VisualDSP++® 5.0
Loader and Utilities Manual

Revision 2.2, March 2009

Part Number
82-000450-01

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

ANALOG DEVICES

Copyright Information

© 2009 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

The Analog Devices logo, VisualDSP++, Blackfin, SHARC, and TigerSHARC are registered trademarks of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.

CONTENTS

PREFACE

Purpose of This Manual xiii
Intended Audience xiii
Manual Contents xiv
What's New in This Manual xiv
Technical or Customer Support xv
Supported Processors xvi
Product Information xvi
Analog Devices Web Site xvi
VisualDSP++ Online Documentation xvii
Technical Library CD xviii
Notation Conventions xviii

INTRODUCTION

Definition of Terms 1-2
Program Development Flow 1-7
Compiling and Assembling 1-8
Linking 1-8
Loading, Splitting, or Both 1-9

Non-bootable Files Versus Boot-loadable Files 1-10
Loader Utility Operations 1-11
Splitter Utility Operations 1-12
Boot Modes 1-13
No-Boot Mode 1-13
PROM Boot Mode 1-14
Host Boot Mode 1-14
Boot Kernels 1-15
Boot Streams 1-16
File Searches 1-17

LOADER/SPLITTER FOR ADSP-BF51X/BF52X/BF54X BLACKFIN PROCESSORS

ADSP-BF51x/BF52x/BF54x Processor Booting 2-2
ADSP-BF51x/BF52x/BF54x Processor Loader Guide 2-5
Using Blackfin Loader Command Line 2-6
File Searches 2-7
File Extensions 2-7
ADSP-BF51x/BF52x/BF54x Blackfin Loader Command-Line Switches 2-8
ADSP-BF51x/BF52x/BF54x Multi-DXE Loader Files 2-17
Using VisualDSP++ Loader 2-18
Using VisualDSP++ Second-Stage Loader 2-20
Using VisualDSP++ ROM Splitter 2-22

LOADER/SPLITTER FOR ADSP-BF53X/BF561 BLACKFIN

PROCESSORS

ADSP-BF53x/BF561 Processor Booting 3-2
ADSP-BF531/BF532/BF533/BF534/BF536/BF537/ BF538/BF539 Processor Booting 3-3
ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processor On-Chip Boot ROM 3-7
ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processor Boot Streams 3-9
ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Block Headers and Flags 3-10
ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Initialization Blocks 3-13
ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 and ADSP-BF535 Processor No-Boot Mode 3-16
ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processor Memory Ranges 3-19
ADSP-BF535 Processor Booting 3-21
ADSP-BF535 Processor On-Chip Boot ROM 3-23
ADSP-BF535 Processor Second-Stage Loader 3-24
ADSP-BF535 Processor Boot Streams 3-27
Loader Files Without a Second-Stage Loader 3-28
Loader Files With a Second-Stage Loader 3-29
Global Headers 3-31
Block Headers and Flags 3-32
ADSP-BF535 Processor Memory Ranges 3-33
Second-Stage Loader Restrictions 3-34

ADSP-BF561 Processor Booting 3-35
ADSP-BF561 Processor On-Chip Boot ROM 3-37
ADSP-BF561 Processor Boot Streams 3-37
ADSP-BF561 Processor Initialization Blocks 3-43
ADSP-BF561 Dual-Core Application Management 3-44
ADSP-BF561 Processor Memory Ranges 3-45
ADSP-BF53x and ADSP-BF561 Multi-Application (Multi-DXE) Management 3-46
ADSP-BF531/BF532/BF533/BF534/BF536/BF537 Processor Compression Support 3-49
Compressed Streams 3-50
Compressed Block Headers 3-51
Uncompressed Streams 3-53
Booting Compressed Streams 3-54
Decompression Initialization Files 3-54
ADSP-BF53x/BF561 Processor Loader Guide 3-57
Using Blackfin Loader Command Line 3-58
File Searches 3-59
File Extensions 3-59
Blackfin Loader Command-Line Switches 3-60
Using VisualDSP++ Loader 3-72
Using VisualDSP++ Compression 3-75
Using VisualDSP++ Second-Stage Loader for ADSP-BF535 Processors 3-76
Using VisualDSP++ ROM Splitter 3-78

ADSP-BF535 and ADSP-BF531/BF532/BF533/BF534/
BF536/BF537/BF538/BF539 Processor No-Boot Mode .. 3-79

LOADER FOR ADSP-2106X/21160 SHARC PROCESSORS

ADSP-2106x/21160 Processor Booting 4-2
Power-Up Booting Process 4-3
Boot Mode Selection 4-5
ADSP-2106x/21160 Boot Modes 4-7
EPROM Boot Mode 4-7
Host Boot Mode 4-11
Link Port Boot Mode 4-15
No-Boot Mode 4-16
ADSP-2106x/21160 Boot Kernels 4-16
ADSP-2106x/21160 Processor Boot Steams 4-17
Boot Kernel Modification and Loader Issues 4-19
ADSP-2106x/21160 Interrupt Vector Table 4-22
ADSP-2106x/21160 Multi-Application (Multi-DXE) Management
4-23
ADSP-2106x/21160 Processor ID Numbers 4-24
ADSP-2106x/21160 Processor Loader Guide 4-25
Using ADSP-2106x/21160 Loader Command Line 4-26
File Searches 4-27
File Extensions 4-27
ADSP-2106x/21160 Loader Command-Line Switches 4-28
Using VisualDSP++ Interface (Load Page) 4-32

LOADER FOR ADSP-21161 SHARC PROCESSORS

ADSP-21161 Processor Booting 5-2
Power-Up Booting Process 5-3
Boot Mode Selection 5-4
ADSP-21161 Processor Boot Modes 5-5
EPROM Boot Mode 5-5
Host Boot Mode 5-9
Link Port Boot Mode 5-12
SPI Port Boot Mode 5-14
No-Boot Mode 5-16
ADSP-21161 Processor Boot Kernels 5-16
ADSP-21161 Processor Boot Streams 5-17
Boot Kernel Modification and Loader Issues 5-18
Rebuilding a Boot Kernel File 5-18
Rebuilding a Boot Kernel Using Command Lines 5-19
Loader File Issues 5-20
ADSP-21161 Processor Interrupt Vector Table 5-21
ADSP-21161 Multi-Application (Multi-DXE) Management .. 5-21
Boot From a Single EPROM 5-22
Sequential EPROM Boot 5-22
Processor ID Numbers 5-23
ADSP-21161 Processor Loader Guide 5-24
Using ADSP-21161 Loader Command Line 5-25
File Searches 5-27

File Extensions 5-27
Loader Command-Line Switches 5-28
Using VisualDSP++ Interface (Load Page) 5-32

LOADER FOR ADSP-2126X/2136X/2137X/2146X SHARC PROCESSORS

ADSP-2126x/2136x/2137x/2146x Processor Booting 6-2
Power-Up Booting Process 6-3
Boot Mode Selection 6-4
ADSP-2126x/2136x/2137x/2146x Processors Boot Modes 6-5
PROM Boot Mode 6-5
Packing Options for External Memory 6-7
Packing and Padding Details 6-7
SPI Port Boot Modes 6-7
SPI Slave Boot Mode 6-8
SPI Master Boot Modes 6-10
Booting From an SPI Flash 6-16
Booting From an SPI PROM (16-bit address) 6-16
Booting From an SPI Host Processor 6-17
Internal Boot Mode 6-17
ADSP-2126x/2136x/2137x/2146x Processors Boot Kernels 6-19
Boot Kernel Modification and Loader Issues 6-20
Rebuilding a Boot Kernel File 6-20
Rebuilding a Boot Kernel Using Command Lines 6-21
Loader File Issues 6-21

ADSP-2126x/2136x/2137x/2146x Processors Interrupt Vector Table 6-22
ADSP-2126x/2136x/2137x/2146x Processor Boot Streams 6-23
ADSP-2126x/2136x/2137x/2146x Processor Block Tags 6-23
INIT_L48 Blocks 6-26
INIT_L16 Blocks 6-27
INIT_L64 Blocks 6-28
FINAL_INIT Blocks 6-28
ADSP-2136x/2137x/2146x Multi-Application (Multi-DXE) Management 6-33
ADSP-2126x/2136x/2137x/2146x Processors Compression Support 6-35
Compressed Streams 6-36
Compressed Block Headers 6-37
Uncompressed Streams 6-38
Overlay Compression 6-39
Booting Compressed Streams 6-39
Decompression Kernel File 6-40
ADSP-2126x/2136x/2137x/2146x Processor Loader Guide 6-41
Using ADSP-2126x/2136x/2137x/2146x Loader Command Line 6-42
File Searches 6-43
File Extensions 6-43
Loader Command-Line Switches 6-44
Using VisualDSP++ Interface (Load Page) 6-49

LOADER FOR TIGERSHARC PROCESSORS

TigerSHARC Processor Booting 7-2
Boot Type Selection 7-3
TigerSHARC Processor Boot Kernels 7-4
Boot Kernel Modification 7-5
TigerSHARC Loader Guide 7-5
Using TigerSHARC Loader Command Line 7-6
File Searches 7-8
File Extensions 7-8
TigerSHARC Command-Line Switches 7-9
Using VisualDSP++ Interface (Load Page) 7-12

SPLITTER FOR SHARC AND TIGERSHARC PROCESSORS

Splitter Command Line 8-2
File Searches 8-3
Output File Extensions 8-4
Splitter Command-Line Switches 8-5
VisualDSP++ Interface (Split Page) 8-9

FILE FORMATS

Source Files A-1
C/C++ Source Files A-2
Assembly Source Files A-2
Assembly Initialization Data Files A-2

Header Files A-3
Linker Description Files A-4
Linker Command-Line Files A-4
Build Files A-4
Assembler Object Files A-5
Library Files A-5
Linker Output Files A-5
Memory Map Files A-6
Loader Output Files in Intel Hex-32 Format A-6
Loader Output Files in Include Format A-8
Loader Output Files in Binary Format A-9
Output Files in Motorola S-Record Format A-10
Splitter Output Files in Intel Hex-32 Format A-12
Splitter Output Files in Byte-Stacked Format A-12
Splitter Output Files in ASCII Format A-14
Debugger Files A-15
Format References A-16

UTILITIES

hexutil – Hex-32 to S-Record File Converter B-2
elf2flt – ELF to BFLT File Converter B-3
fltdump – BFLT File Dumper B-4

INDEX

PREFACE

Thank you for purchasing Analog Devices, Inc. development software for Analog Devices embedded processors.

Purpose of This Manual

The *VisualDSP++ 5.0 Loader and Utilities Manual* contains information about the loader/splitter program for Analog Devices processors.

The manual describes the loader/splitter operations for these processors and references information about related development software. It also provides information about the loader and splitter command-line interfaces.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. The manual assumes the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual, but should supplement it with other texts, such as hardware reference and programming reference manuals, that describe their target architecture.

Manual Contents

The manual contains:

- Chapter 1, "Introduction"
- Chapter 2, "Loader/Splitter for ADSP-BF51x/BF52x/BF54x Blackfin Processors"
- Chapter 3, "Loader/Splitter for ADSP-BF53x/BF561 Blackfin Processors"
- Chapter 4, "Loader for ADSP-2106x/21160 SHARC Processors"
- Chapter 5, "Loader for ADSP-21161 SHARC Processors"
- Chapter 6, "Loader for ADSP-2126x/2136x/2137x/2146x SHARC Processors"
- Chapter 7, "Loader for TigerSHARC Processors"
- Chapter 8, "Splitter for SHARC and TigerSHARC Processors"
- Appendix A, "File Formats"
- Appendix B, "Utilities"

What's New in This Manual

This revision of the *VisualDSP++ 5.0 Loader and Utilities Manual* documents loader and splitter functionality that is new to VisualDSP++ 5.0 and updates (up to update 6), including support for new SHARC processors.

In addition, modifications and corrections based on errata reports against the previous revision of the manual have been made.

Technical or Customer Support

You can reach Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at
  http://www.analog.com/processors/technical_support
- E-mail tools questions to
  processor.tools.support@analog.com
- E-mail processor questions to
  processor.support@analog.com (World wide support)
  processor.europe@analog.com (Europe support)
  processor.china@analog.com (China support)
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:
  Analog Devices, Inc.
  One Technology Way
  P.O. Box 9106
  Norwood, MA 02062-9106
  USA

Supported Processors

Loader and utility programs of VisualDSP++ 5.0 supports the following Analog Devices, Inc. processors.

- Blackfin® (ADSP-BFxxx)
- SHARC® (ADSP-21xxx)
- TigerSHARC® (ADSP-TSxxx)

The majority of the information in this manual applies to all processors. For a complete list of processors supported by VisualDSP++ 5.0, refer to the online Help.

Product Information

Product information can be obtained from the Analog Devices Web site, VisualDSP++ online Help system, and a technical library CD.

Analog Devices Web Site

The Analog Devices Web site, www.analog.com, provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

To access a complete technical library for each processor family, go to http://www.analog.com/processors/technical_library. The manuals selection opens a list of current manuals related to the product as well as a link to the previous revisions of the manuals. When locating your manual title, note a possible errata check mark next to the title that leads to the current correction report against the manual.

Also note, MyAnalog.com is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information about products you are interested in. You can choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests, including documentation errata against all manuals. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Visit MyAnalog.com to sign up. If you are a registered user, just log on. Your user name is your e-mail address.

VisualDSP++ Online Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, Dinkum Abridged C++ library, and FLEXnet License Tools software documentation. You can search easily across the entire VisualDSP++ documentation set for any topic of interest.

For easy printing, supplementary Portable Documentation Format (.pdf) files for all manuals are provided on the VisualDSP++ installation CD.

Each documentation file type is described as follows.

File	Description
.chm	Help system files and manuals in Microsoft help format
.htm or .html	Dinkum Abridged C++ library and FLEXnet License Tools software documentation. Viewing and printing the .html files requires a browser, such as Internet Explorer 6.0 (or higher).
.pdf	VisualDSP++ and processor manuals in PDF format. Viewing and printing the .pdf files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

Technical Library CD

The technical library CD contains seminar materials, product highlights, a selection guide, and documentation files of processor manuals, VisualDSP++ software manuals, and hardware tools manuals for the following processor families: Blackfin, SHARC, TigerSHARC, ADSP-218x, and ADSP-219x.

To order the technical library CD, go to http://www.analog.com/processors/technical_library, navigate to the manuals page for your processor, click the request CD check mark, and fill out the order form.

Data sheets, which can be downloaded from the Analog Devices Web site, change rapidly, and therefore are not included on the technical library CD. Technical manuals change periodically. Check the Web site for the latest manual revisions and associated documentation errata.

Notation Conventions

Text conventions used in this manual are identified and described as follows. Additional conventions, which apply only to specific chapters, may appear throughout this document.

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
	Note: For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	Caution: Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
	Warning: Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for the devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

1 INTRODUCTION

The majority of this manual describes the loader utility (or loader) program as well as the process of loading and splitting, the final phase of the application development flow.

Most of this chapter applies to all 8-, 16-, and 32-bit processors. Information specific to a particular processor, or to a particular processor family, is provided in the following chapter.

- Chapter 2, "Loader/Splitter for ADSP-BF51x/BF52x/BF54x Blackfin Processors"
- Chapter 3, "Loader/Splitter for ADSP-BF53x/BF561 Blackfin Processors"
- Chapter 4, "Loader for ADSP-2106x/21160 SHARC Processors"
- Chapter 5, "Loader for ADSP-21161 SHARC Processors"
- Chapter 6, "Loader for ADSP-2126x/2136x/2137x/2146x SHARC Processors"
- Chapter 7, "Loader for TigerSHARC Processors"
- Chapter 8, "Splitter for SHARC and TigerSHARC Processors"
- Appendix A, "File Formats"
- Appendix B, "Utilities"

The code examples in this manual have been compiled using VisualDSP++ 5.0. The examples compiled with another version of VisualDSP++ may result in build errors or different output; although, the highlighted algorithms stand and should continue to stand in future releases of VisualDSP++.

Definition of Terms

Loader and Loader Utility

The term *loader* refers to a *loader utility* that is part of the VisualDSP++ development tools suite. The loader utility post-processes one or multiple executable (`.dxe`) files, extracts segments that have been declared by the `TYPE(RAM)` command in a Linker Description File (`.ldf`), and generates a loader file (`.ldr`). Since the `.dxe` file meets the Executable and Linkable Format (ELF) standard, the loader utility is often called `elfloader` utility. See also "Loader Utility Operations" on page 1-11.

Splitter Utility

The *splitter utility* is part of the VisualDSP++ development tools suite. The splitter utility post-processes one or multiple executable (`.dxe`) files, extracts segments that have been declared by the `TYPE(ROM)` command a Linker Description File (`.ldf`), and generates a file consisting of processor instructions (opcodes). If burned into an EPROM or flash memory device which connects to the target processor's system bus, the processor can directly fetch and execute these instructions. See also "Splitter Utility Operations" on page 1-12.

Splitter and loader jobs can be managed either by separate utility programs or by the same program (see "Non-bootable Files Versus Boot-loadable Files" on page 1-10). In the later case, the generated output file may contain code instructions and boot streams.

Loader File

A *loader file* is generated by the loader utility. The file typically has the `.ldr` extension and is often called an LDR file. Loader files can meet one of multiple formats. Common formats are Intel hex-32, binary, or ASCII representation. Regardless of the format, the loader file describes a boot image, which can be seen as the binary version of the loader file. See also "Non-bootable Files Versus Boot-loadable Files" on page 1-10.

Loader Command Line

If invoked from a command-line prompt, the loader and splitter utilities accept numerous control switches to customize the loader file generation.

Loader Property Page

The *loader property page* is part of the **Project Options** dialog box of the VisualDSP++ graphical user interface. The property page is a graphical tool that assists in composing the loader utility's command line.

Boot Mode

Most processors support multiple boot modes. A *boot mode* is determined by special input pins that are interrogated when the processor awakes from either a reset or power-down state. See also "Boot Modes" on page 1-13.

Boot Kernel

A *boot kernel* is software that runs on the target processor. It reads data from the boot source and interprets the data as defined in the boot stream format. The boot kernel can reside in an on-chip boot ROM or in an off-chip ROM device. Often, the kernel has to be pre-booted from the boot source before it can be executed. In this case, the loader utility puts a default kernel to the front of the boot image, or, allows the user to specify a customized kernel. See also "Boot Kernels" on page 1-15.

Boot ROM

A *boot ROM* is an on-chip read-only memory that holds the boot kernel and, in some cases, additional advanced booting routines.

Second-Stage Loader

A *second-stage loader* is a special boot kernel that extends the default booting mechanisms of the processor. It is typically booted by a first-stage kernel in a standard boot mode configuration. Afterward, it executes and boots in the final applications. See also "Boot Kernels" on page 1-15.

Boot Source

A *boot source* refers to the interface through which the boot data is loaded as well as to the storage location of a boot image, such as a memory or host device.

Boot Image

A *boot image* that can be seen as the binary version of a loader file. Usually, it has to be stored into a physical memory that is accessible by either the target processor or its host device. Often it is burned into an EPROM or downloaded into a flash memory device using the VisualDSP++ Flash Programmer plug-in.

The boot image is organized in a special manner required by the boot kernel. This format is called a boot stream. A boot image can contain one or multiple boot streams. Sometimes the boot kernel itself is part of the boot image.

Boot Stream

A *boot stream* is basically a list of boot blocks. It is the data structure that is processed and interpret by the boot kernel. The VisualDSP++ loader utility generates loader files that contain one or multiple boot streams. A boot

stream often represents one application. However, a linked list of multiple application-level boot streams is referred to as a boot stream. See also "Boot Streams" on page 1-16.

Boot Host

A *boot host* is a processor or programmable logic that feeds the device configured in a slave boot mode with a boot image or a boot stream.

Boot Block

Multiple *boot blocks* form a boot stream. These blocks consist of boot data that is preceded by a block header. The header instructs the boot kernel how to interpret the payload data. In some cases, the header may contain special instructions only. In such blocks, there is likely no payload data present.

Boot Code

Boot code is used to refer to all boot relevant ROM code. Boot code typically consists of the pre-boot routine and the boot kernel.

Boot Strapping

If the boot process consists of multiple steps, such as pre-loading the boot kernel or managing second-stage loaders, this is called *boot strapping* or boot ROM.

Initialization Code

Initialization code or *initcode* is part of a boot stream for Blackfin processors and can be seen as a special boot block. While normally all boot blocks of an application are booted in first and control is passed to the application afterward, the initialization code executes at boot time. It is common that an initialization code is booted and executed before any other boot block. This initialization code can customize the target system for optimized boot processing.

Global Header

Some boot kernels expect a boot stream to be headed by a special information tag. The tag is referred to as a *global header*.

Callback Routine

Some processors can optionally call a user-defined routine after a boot block has been loaded and processed. This is referred to as a *callback routine*. It provides hooks to implement checksum and decompression strategies.

Slave Boot

The term *slave boot* spawns all boot modes where the target processor functions as a slave. This is typically the case when a host device loads data into the target processor's memories. The target processor can wait passively in idle mode or support the host-controlled data transfers actively. Note that the term host boot usually refers only to boot modes that are based on so-called host port interfaces.

Master Boot

The term *master boot* spawns all boot modes where the target processor functions as master. This is typically the case when the target processor reads the boot data from parallel or serial memories.

Boot Manager

A *boot manager* is a firmware that decides what application has to be booted. An application is usually represented by a VisualDSP++ project and stored in a `.dxe` file. The boot manger itself can be managed within an application `.dxe` file, or have its own separate `.dxe` file. Often, the boot manager is executed by so-called initialization codes.

In slave boot scenarios, boot management is up to the host device and does not require special VisualDSP++ support.

Multi-.dxe Boot

A loader file may can contain data of multiple application (`.dxe`) files if the loader utility was invoked by specifying multiple `.dxe` files. Either a boot manager decides what application has to be booted exclusively or, alternatively, one application can terminate and initiate the next application to be booted. In some cases, a single application can also consist of multiple `.dxe` files.

Next .dxe File Pointer

If a loader file contains multiple applications, some boot stream formats enable them to be organized as a linked list. The next `.dxe` pointer (NDP) is simply a pointer to a location where the next application's boot stream resides.

Preboot Routine

A preboot routine is present in the boot ROM of parts that feature OTP memory on a processor. Preboot reads OTP memory and customizes several MMR registers based on factory and user instructions, as programmed to OTP memory. A preboot routine executes prior to the boot kernel.

Program Development Flow

Figure 1-1 is a simplified view of the application development flow.

The development flow can be split into three phases:

1. Compiling and Assembling
2. Linking
3. Loading, Splitting, or Both

A brief description of each phase follows.

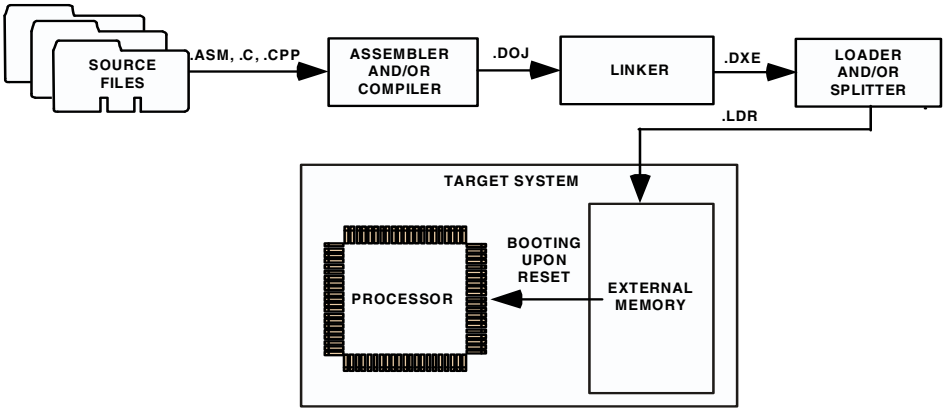

Figure 1-1. Program Development Flow

Compiling and Assembling

Input source files are compiled and assembled to yield object files. Source files are text files containing C/C++ code, compiler directives, possibly a mixture of assembly code and directives, and, typically, preprocessor commands. The assembler and compiler are documented in the *VisualDSP++ 5.0 Assembler and Preprocessor Manual* and *VisualDSP++ 5.0 C/C++ Compiler and Library Manual*, which are part of the online help.

Linking

Under the direction of the linker description file (LDF) and linker settings, the linker consumes separately-assembled object and library files to yield an executable file. If specified, the linker also produces the shared memory files and overlay files. The linker output (`.dxe` files) conforms to the ELF standard, an industry-standard format for executable files. The linker also produces map files and other embedded information (DWARF-2) used by the debugger.

These executable files are not readable by the processor hardware directly. They are neither supposed to be burned onto an EPROM or flash memory device. Executable files are intended for VisualDSP++ debugging targets, such as the simulator or emulator. Refer to the *VisualDSP++ 5.0 Linker and Utilities Manual* and online Help for information about linking and debugging.

Loading, Splitting, or Both

Upon completing the debug cycle, the processor hardware needs to run on its own, without any debugging tools connected. After power-up, the processor's on-chip and off-chip memories need to be initialized. The process of initializing memories is often referred to as *booting*. Therefore, the linker output must be transformed to a format readable by the processor. This process is handled by the loader and/or splitter utility. The loader/splitter utility uses the debugged and tested executable files as well as shared memory and overlay files as inputs to yield a processor-loadable file.

VisualDSP++ 5.0 includes these loader and splitter utilities:

- `elfloader.exe` (loader utility) for Blackfin, TigerSHARC, and SHARC processors. The loader utility for Blackfin processors also acts as a ROM splitter utility when evoked with the corresponding switches.
- `elfspl21k.exe` (ROM splitter utility) for TigerSHARC and SHARC processors.

The loader/splitter output is either a boot-loadable or non-bootable file. The output is meant to be loaded onto the target. There are several ways to use the output:

- Download the loadable file into the processor's PROM space on an EZ-KIT Lite® board via the Flash Programmer plug-in. Refer to VisualDSP++ Help for information on the Flash Programmer.
- Use VisualDSP++ to simulate booting in a simulator session (currently supported on the ADSP-21060, ADSP-21061, ADSP-21065L, ADSP-21160, and ADSP-21161 processors). Load the loader file and then reset the processor to debug the booting routines. No hardware is required: just point to the location of the loader file, letting the simulator to do the rest. You can step through the boot kernel code as it brings the rest of the code into memory.
- Store the loader file in an array for a multiprocessor system. A master (host) processor has the array in its memory, allowing a full control to reset and load the file into the memory of a slave processor.

Non-bootable Files Versus Boot-loadable Files

A non-bootable file executes from an external memory of the processor, while a boot-loadable file is transported into and executes from an internal memory of the processor. The boot-loadable file is then programmed into an external memory device (burned into EPROM) within your target system. The loader utility outputs loadable files in formats readable by most EPROM burners, such as Intel hex-32 and Motorola S formats. For advanced usage, other file formats and boot modes are supported.
See "File Formats" on page A-1.

A non-bootable EPROM image file executes from an external memory of the processor, bypassing the built-in boot mechanisms. Preparing a non-bootable EPROM image is called *splitting*. In most cases (except for

Blackfin processors), developers working with floating- and fixed-point processors use the splitter instead of the loader utility to produce a non-bootable memory image file.

A booting sequence of the processor and application program design dictate the way loader/splitter utility is called to consume and transform executable files:

- For Blackfin processors, loader and splitter operations are handled by the loader utility program, `elfloader.exe`. The splitter is invoked by a different set of command-line switches than the loader.

  In VisualDSP++ 5.0, with the addition of the `-readall` switch, the loader utility for the ADSP-BF51x/BF52x/BF54x Blackfin processors can call the splitter program automatically. For more information, see -readall #.

- For TigerSHARC and SHARC processors, splitter operations are handled by the splitter program, `elfspl21k.exe`.

Loader Utility Operations

Common tasks performed by the loader utility can include:

- Processing the loader option settings or command-line switches.
- Formatting the output `.ldr` file according to user specifications. Supported formats are binary, ASCII, Intel hex-32, and more. Valid file formats are described in "File Formats" on page A-1.
- Packing the code for a particular data format: 8-, 16- or 32-bit for some processors.
- Adding the code and data from a specified initialization executable file to the loader file, if applicable.
- Adding a boot kernel on top of the user code.

- If specified, preprogramming the location of the `.ldr` file in a specified PROM space.
- Specifying processor IDs for multiple input `.dxe` files for a multiprocessor system, if applicable.

You can run the loader utility from the VisualDSP++ Integrated Development and Development Environment (IDDE), when the IDDE is available, or from the command line. In order to do so in the IDDE, open the **Project Options** dialog box from the **Project** menu, and change the project's target type from **Executable file** to **Loader File**.

Loader utility operations depend on the loader options, which control how the loader utility processes executable files into boot-loadable files, letting you select features such as kernels, boot modes, and output file formats. These options are set on the **Load** pages of the **Project Options** dialog box in the IDDE or on the loader command line. Option settings on the **Load** pages correspond to switches typed on the `elfloader.exe` command line.

Splitter Utility Operations

Splitter utility operations depend on the splitter options, which control how the splitter utility processes executable files into non-bootable files:

- For Blackfin processor, the loader utility includes the ROM splitter capabilities invoked through the **Project Options** dialog box. Refer to "Using VisualDSP++ ROM Splitter" on page 3-78. Option settings in the dialog box correspond to switches typed on the `elfloader.exe` command line.
- For SHARC and TigerSHARC processors, change the project's target type to **Splitter file**. The splitter options are set via the **Project: Split** page of the **Project Options** dialog box. Refer to "Splitter for

SHARC and TigerSHARC Processors" on page 8-1. Option settings in the dialog box correspond to switches typed on the `elfspl21k.exe` command line.

Boot Modes

Once an executable file is fully debugged, the loader utility is ready to convert the executable file into a processor-loadable (boot-loadable) file. The loadable file can be automatically downloaded (booted) to the processor after power-up or after a software reset. The way the loader utility creates a boot-loadable file depends upon how the loadable file is booted into the processor.

The boot mode of the processor is determined by sampling one or more of the input flag pins. Booting sequences, highly processor-specific, are detailed in the following chapters.

Analog Devices processors support different boot mechanisms. In general, the following schemes can be used to provide program instructions to the processors after reset.

- No-Boot Mode
- PROM Boot Mode
- Host Boot Mode

No-Boot Mode

After reset, the processor starts fetching and executing instructions from EPROM/flash memory devices directly. This scheme does not require any loader mechanism. It is up to the user program to initialize volatile memories.

The splitter utility generates a file that can be burned into the PROM memory.

PROM Boot Mode

After reset, the processor starts reading data from a parallel or serial PROM device. The PROM stores a formatted boot stream rather than raw instruction code. Beside application data, the boot stream contains additional data, such as destination addresses and word counts. A small program called a boot kernel (described on page 1-15) parses the boot stream and initializes memories accordingly. The boot kernel runs on the target processor. Depending on the architecture, the boot kernel may execute from on-chip boot RAM or may be preloaded from the PROM device into on-chip SRAM and execute from there.

The loader utility generates the boot stream from the linker output (an executable file) and stores it to file format that can be burned into the PROM.

Host Boot Mode

In this scheme, the target processor is a slave to a host system. After reset, the processor delays program execution until the slave gets signalled by the host system that the boot process has completed. Depending on hardware capabilities, there are two different methods of host booting. In the first case, the host system has full control over all target memories. The host halts the target while initializing all memories as required. In the second case, the host communicates by a certain handshake with the boot kernel running on the target processor. This kernel may execute from on-chip ROM or may be preloaded by the host devices into the processor's SRAM by any bootstrapping scheme.

The loader/splitter utility generates a file that can be consumed by the host device. It depends on the intelligence of the host device and on the target architecture whether the host expects raw application data or a formatted boot stream.

In this context, a boot-loadable file differs from a non-bootable file in that it stores instruction code in a formatted manner in order to be processed by a boot kernel. A non-bootable file stores raw instruction code.

Boot Kernels

A boot kernel refers to the resident program in the boot ROM space responsible for booting the processor. Alternatively (or in absence of the boot ROM), the boot kernel can be preloaded from the boot source by a bootstrapping scheme.

When a reset signal is sent to the processor, the processor starts booting from a PROM, host device, or through a communication port. For example, an ADSP-2106x/2116x processor, brings a 256-word program into internal memory for execution. This small program is a boot kernel.

The boot kernel then brings the rest of the application code into the processor's memory. Finally, the boot kernel overwrites itself with the final block of application code and jumps to the beginning of the application program.

Some of the newer Blackfin processors do not require to load a boot kernel—a kernel is already present in the on-chip boot ROM. It allows the entire application program's body to be booted into the internal and external memories of the processor. The boot kernel in the on-chip ROM behaves similar to the second-stage loader of the ADSP-BF535 processors. The boot ROM has the capability to parse address and count information for each bootable block.

Boot Streams

The loader utility's output (`.ldr` file) is essentially the same executable code as in the input `.dxe` file; the loader utility simply repackages the executable as shown in Figure 1-2.

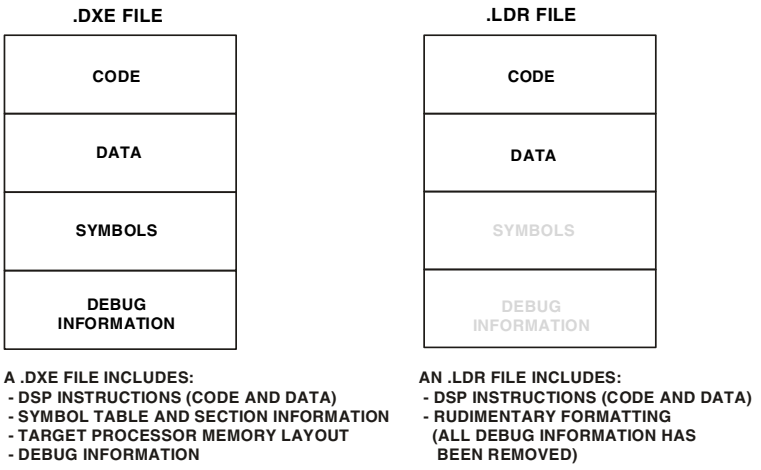

Figure 1-2. A .dxe File Versus an .ldr File

Processor code and data in a loader file (also called a boot stream) is split into blocks. Each code block is marked with a tag that contains information about the block, such as the number of words and destination in the processor's memory. Depending on the processor family, there can be additional information in the tag. Common block types are "zero" (memory is filled with `0`s); nonzero (code or data); and final (code or data). Depending on the processor family, there can be other block types.

Refer to the following chapters to learn more about boot streams.

File Searches

File searches are important in the loader utility operation. The loader utility supports relative and absolute directory names and default directories. File searches occur as follows.

- Specified path—If relative or absolute path information is included in a file name, the loader utility searches only in that location for the file.
- Default directory—If path information is not included in the file name, the loader utility searches for the file in the current working directory.
- Overlay and shared memory files—The loader utility recognizes overlay and shared memory files but does not expect these files on the command line. Place the files in the directory that contains the executable file that refers to them, or place them in the current working directory. The loader utility can locate them when processing the executable file.

When providing an input or output file name as a loader/splitter command-line parameter, use these guidelines:

- Enclose long file names within straight quotes, "`long file name`".
- Append the appropriate file extension to each file.

2 LOADER/SPLITTER FOR ADSP-BF51X/BF52X/BF54X BLACKFIN PROCESSORS

This chapter explains how the loader/splitter utility (`elfloader.exe`) is used to convert executable (`.dxe`) files into boot-loadable or non-bootable files for the ADSP-BF51x, ADSP-BF52x, and ADSP-BF54x Blackfin processors.

Refer to "Introduction" on page 1-1 for the loader utility overview. Loader operations specific to the ADSP-BF51x/BF52x/BF54x Blackfin processors are detailed in the following sections.

- "ADSP-BF51x/BF52x/BF54x Processor Booting" on page 2-2
  Provides general information on various boot modes, including information on second-stage kernels.
- "ADSP-BF51x/BF52x/BF54x Processor Loader Guide" on page 2-5
  Provides reference information on the loader utility's command-line syntax and switches.

ADSP-BF51x/BF52x/BF54x Processor Booting

Table 2-1 lists the part numbers that currently comprise the ADSP-BF51x/BF52x/BF54x families of Blackfin processors. Future releases of VisualDSP++ may support additional processors.

Upon reset, an ADSP-BF51x/BF52x/BF54x processor starts fetching and executing instructions from the on-chip boot ROM at address 0xEF00 0000. The boot ROM is an on-chip read-only memory that holds a boot kernel program to load data from an external memory or host device. The boot ROM details can be found in the corresponding *Blackfin Processor Hardware Reference*.

Table 2-1. ADSP-BF51x/BF52x/BF54x Part Numbers

Processor Family	Part Numbers
ADSP-BF518	ADSP-BF512, ADSP-BF514, ADSP-BF516, ADSP-BF518
ADSP-BF526	ADSP-BF522, ADSP-BF524, ADSP-BF526
ADSP-BF527	ADSP-BF523, ADSP-BF525, ADSP-BF527
ADSP-BF548	ADSP-BF542, ADSP-BF544, ADSP-BF547, ADSP-BF548, ADSP-BF549
ADSP-BF548M	ADSP-BF542M, ADSP-BF544M, ADSP-BF547M, ADSP-BF548M, ADSP-BF549M

There are other boot modes available, including idle (no-boot) mode. The processor transitions into the boot mode sequence configured by the BMODE pins; see Table 2-2 and Table 2-3. The BMODE pins are dedicated mode-control pins; that is, no other functions are performed by the pins. The pins can be read through bits in the system configuration register (SYSCR).

There are two categories of boot modes: master and slave. In master boot modes, the processor actively *loads data* from parallel or serial memory devices. In slave boot modes, the processor *receives data* from parallel or serial memory devices.

Refer to the processor's data sheet and hardware reference manual for more information on system configuration, peripherals, registers, and operating modes:

- Blackfin processor data sheets can be found at http://www.analog.com/en/embedded-processing-dsp/blackfin/processors/data-sheets/resources/index.html.
- Blackfin processor manuals can be found at http://www.analog.com/en/embedded-processing-dsp/blackfin/processors/manuals/resources/index.html.

Table 2-2. ADSP-BF51x Boot Modes

Boot Source	BMODE[2:0]	Start Address
Idle (no-boot)	000	N/A
Eight- or 16-bit external flash memory (default mode)	001	0x2000 0000
Internal SPI memory	010	0x2030 0000
External SPI memory (EEPROM or flash)	011	0x0000 0000
SPI0 host device	100	N/A
One-time programmable (OTP) memory	101	N/A
SDRAM memory	110	N/A
UART0 host	111	N/A

Table 2-3. ADSP-BF52x/BF54x Boot Modes

Boot Source	BMODE[3:0]	Start Address
Idle (no-boot)	0000	N/A
Eight- or 16-bit external flash memory (default mode)	0001	0x2000 0000
16-bit asynchronous FIFO	0010	0x2030 0000
Eight-, 16-, 24-, or 32-bit addressable SPI memory	0011	0x0000 0000
External SPI host device	0100	N/A
Serial TWI memory	0101	0x0000 0000
TWI host	0110	N/A
UART0 host on ADSP-BF52x processors; UART1 host on ADSP-BF54x processors	0111	N/A
UART1 host on ADSP-BF52x processors; Reserved on ADSP-BF54x processors	1000	N/A
Reserved	1001	N/A
SDRAM/DDR	1010	0x0000 0010
OTP memory	1011	default page 0x40
Eight- or 16-bit NAND flash memory	1100, 1101	0x0000 0000
16-bit host DMA	1110	N/A
Eight-bit host DMA	1111	N/A

ADSP-BF51x/BF52x/BF54x Processor Loader Guide

The loader utility post processes VisualDSP++ executable (`.dxe`) files and generates loader (`.ldr`) files. A loader file can be formatted as binary, ASCII or Intel hex style. An `.ldr` file contains the boot stream in a format expected by the on-chip boot kernel.

Loader utility operations depend on the loader options, which control how the utility processes executable files. You select features such as boot modes, boot kernels, and output file formats via the options. The options are specified on the loader utility's command line or via the **Load** page of the **Project Options** dialog box in the VisualDSP++ environment. The **Load** page consists of multiple panes. When you open the **Load** page, the default loader settings for the selected processor are set already.

Option settings on the **Load** page correspond to switches displayed on the command line.

These sections describe how to produce a bootable (single and multiple) or non-bootable loader file:

- "Using Blackfin Loader Command Line" on page 2-6
- "Using VisualDSP++ Loader" on page 2-18
- "Using VisualDSP++ Second-Stage Loader" on page 2-20
- "Using VisualDSP++ ROM Splitter" on page 2-22

Using Blackfin Loader Command Line

The ADSP-BF51x/BF52x/BF54x Blackfin loader utility uses the following command-line syntax.

For a single input file:

```
elfloader inputfile -proc processor [-switch …]
```

For multiple input files:

```
elfloader inputfile1 inputfile2 … -proc processor [-switch …]
```

where:

- *inputfile*—Name of the executable (`.dxe`) file to be processed into a single boot-loadable or non-bootable file. An input file name can include the drive and directory. For multiprocessor or multi-input systems, specify multiple input `.dxe` files. Put the input file names in the order in which you want the loader utility to process the files. Enclose long file names within straight quotes, "`long file name`".
- `-proc` *processor*—Part number of the processor (for example, `-proc ADSP-BF542`) for which the loadable file is built. Provide a processor part number for every input `.dxe` if designing multiprocessor systems; see Table 2-1.
- *-switch …*—One or more optional switches to process. Switches select operations and modes for the loader utility.

Command-line switches may be placed on the command line in any order, except the order of input files for a multi-input system. For a multi-input system, the loader utility processes the input files in the order presented on the command line.

File Searches

File searches are important in loader processing. The loader utility supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur as described on page 1-17.

File Extensions

Some loader switches take a file name as an optional parameter. Table 2-4 lists the expected file types, names, and extensions.

Table 2-4. File Extensions

Extension	File Description
`.dxe`	Loader input files, boot kernel files, and initialization files
`.ldr`	Loader output file
`.knl`	Loader output files containing kernel code only when two output files are selected

In some cases, the loader utility expects the overlay input files with the `.ovl` file extension, shared memory input files with the `.sm` extension, or both but does not expect those files to appear on a command line or on the **Load** property page. The loader utility finds these files in the directory of the associated `.dxe` files, in the current working directory, or in the directory specified in the `.ldf` file.

ADSP-BF51x/BF52x/BF54x Blackfin Loader Command-Line Switches

A summary of the ADSP-BF51x/BF52x/BF54x Blackfin loader command-line switches appears in Table 2-5. For a quick on-line help on the switches available for a specific processor; for example an ADSP-BF548 processor, use the following command line.

```
elfloader -proc ADSP-BF548 -help
```

Table 2-5. ADSP-BF51x/BF52x/BF54x Loader Command-Line Switch Summary

Switch	Description

`-b flash`
`-b prom`
`-b spimaster`
`-b spislave`
`-b TWImaster`
`-b TWIslave`
`-b UART`
`-b FIFO`
`-b OTP`
`-b NAND`
	The `-b {prom|flash|spimaster|spislave|UART|TWImaster|TWIslave|FIFO|OTP|NAND}` switch directs the loader utility to prepare a boot-loadable file for the specified boot mode. Valid boot modes include flash, PROM, SPI master, SPI slave, UART, TWI master, TWI slave, FIFO, OTP, NAND, and SDRAM/DDR.
	
	If `-b` does not appear on the command line, the default is `-b flash`.

`-CRC32 [polynomial]`	The `-CRC32` (polynomial coefficient) switch directs the loader utility to generate CRC32 checksum. Use a polynomial coefficient if specified; otherwise, use default `0xD8018001`. This switch inserts an initcode boot block that calls an initialization routine residing in the on-chip boot ROM. The argument field of the boot block provides the used polynomial. The loader utility calculates the CRC checksum for all subsequent data blocks and stores the result in the block header's argument field. The CRC32 checksum is not performed by the ADSP-BF52x boot kernel.

Table 2-5. ADSP-BF51x/BF52x/BF54x Loader Command-Line Switch Summary (Cont'd)

Switch	Description
`-callback sym=`*`symbol`* `[arg=const32]`	The `-callback` switch takes a `sym=`*`symbol`* (no spaces) assignment. The switch directs the loader utility to isolate the named subroutine into a separate block, set the block header's `BFLAG_CALLBACK` flag, and fill in the block header's argument field with the specified constant 32-bit values. The switch is used for boot-time callbacks. The callback is guaranteed to be made prior to the target address of `sym=`*`symbol`*. The `-callback` cannot be used with `-CRC32`.
`-dmawidth #`	The `-dmawidth {8|16|32}`switch specifies a DMA width (in bits) for memory boot modes. It controls the `DMACODE` bit field issued to the boot block headers by the `-width` switch. For FIFO boot mode, 16 is the only DMA width. SPI, TWI, and UART modes use 8-bit DMA.

`-f hex`
`-f ASCII`
`-f binary`
`-f include`
	The `-f {hex|ASCII|binary|include}` switch specifies the format of a boot-loadable file: Intel hex-32, ASCII, binary, or include. If the `-f` switch does not appear on the command line, the default file format is hex for flash/PROM boot modes; and ASCII for other boot modes.

`-h` or `-help`	The `-help` switch invokes the command-line help, outputs a list of command-line switches to standard output, and exits. By default, the `-h` switch alone provides help for the loader driver. To obtain a help screen for your target Blackfin processor, add the `-proc` switch to the command line. For example, type `elfloader -proc ADSP-BF542 -h` to obtain help for the ADSP-BF542 processor.

Table 2-5. ADSP-BF51x/BF52x/BF54x Loader Command-Line Switch Summary (Cont'd)

Switch	Description
`-init filename.dxe`	The `-init filename.dxe` switch directs the loader utility to include the initialization code from the named executable file. The loader utility places the code and data from the initialization sections at the top of the boot stream. The boot kernel loads the code and then calls it. It is the code's responsibility to save/restore state/registers and then perform an RTS back to the kernel. Initcodes can be written in C language and are compliant to C calling conventions. The `-init filename.dxe` switch can be used multiple times to specify the same file or different files a number of times. The loader utility will place the code from the initialization files in the order the files appear on the command line. For more information, see "ADSP-BF51x/BF52x/BF54x Multi-DXE Loader Files" on page 2-17.

Table 2-5. ADSP-BF51x/BF52x/BF54x Loader Command-Line Switch Summary (Cont'd)

Switch	Description
`-initcall sym=sym_symbol at=at_symbol [stride=DstAddrGap count=times]`	While the `-init filename.dxe` switch integrates initialization codes managed by a separate application program, the `-initcall` switch controls calls to initialization subroutines that are part of the same application. The `-initcall` switch directs the loader utility to dispatch a boot-time initialization call to the `sym` subroutine when the `at` symbol is encountered and loaded. The `stride` and `count` parameters are optional: • If an optional `stride=` constant 32-bit value is specified, the loader utility insets the target program call every `stride=` target address locations. • If an optional `count=` constant 32-bit value is specified, the loader utility insets the target program call `count=` times, every `stride=` target address locations apart. A `count` value without a `stride` value is an error. For example, the following command line `-initcall sym=_initcode at=_othersymbol stride=0x100 count=5` results in function `_initcode` being called five times the first time, just prior to data in `_othersymbol` being booted. Thereafter, every 256 destination load addresses `_initcode` is called again until a total of five calls have been made. `-initcall` restrictions: • `-initcall` target (*`sym_symbol`*) must be a routine entry point, end with an RTS. It can be written in C language and can rely on the presence of a stack. However, the routine must not call any libraries, not rely on compiler run-time environment (such as heaps) – must be self-contained • `-initcall` subroutine must be previously loaded and still in memory • `-initcall` subroutine cannot contain any forward references to code not yet loaded • *`sym_symbol`* address must be less than *`at_symbol`* address For more information, see "ADSP-BF51x/BF52x/BF54x Multi-DXE Loader Files" on page 2-17.

Table 2-5. ADSP-BF51x/BF52x/BF54x Loader Command-Line Switch Summary (Cont'd)

Switch	Description

`-kb flash`
`-kb prom`
`-kb spimaster`
`-kb spislave`
`-kb UART`
`-kb TWImaster`
`-kb TWIslave`
`-kb FIFO`
`-kb otp`
`-kb nand`
	The `-kb {flash|prom|spimaster|spislave|UART|TWImaster|TWIslave|FIFO|otp|nand}` switch specifies the boot mode: flash, PROM, SPI master, SPI slave, UART, TWI master, TWI slave, or FIFO, OTP, NAND, and SDRAM/DDR. for the initialization code and/or boot kernel output file if two output loader files are selected.
	 The `-kb` switch must be used in conjunction with the `-o2` switch.
	If the `-kb` switch is absent from the command line, the loader utility generates the file for the init and/or boot kernel code in the same boot mode as used to output the user application program.

`-kf hex`
`-kf ascii`
`-kf binary`
`-kf include`
	The `-kf {hex|ascii|binary|include}` switch specifies the output file format (hex, ASCII, binary, include) for the initialization and/or boot kernel code if two output files from the loader utility are selected: one file for the init code and/or boot kernel and one file for user application code.
	 The `-kf` switch must be used in conjunction with the `-o2` switch.
	If `-kf` is absent from the command line, the loader utility generates the file for the initialization and /or boot kernel code in the same format as for the user application code.

`-kp #`	The `-kp #` switch specifies a hex flash/PROM start address for the initialization and/or boot kernel code. A valid value is between `0x0` and `0xFFFFFFFF`. The specified value is ignored when neither kernel nor initialization code is included in the loader file.

Table 2-5. ADSP-BF51x/BF52x/BF54x Loader Command-Line Switch Summary (Cont'd)

Switch	Description
`-kwidth #`	The `-kwidth {8|16|32}` switch specifies an external memory device width (in bits) for the initialization code and/or the boot kernel if two output files from the loader utility are selected. If `-kwidth` is absent from the command line, the loader utility generates the boot kernel file in the same width as the user application program. The `-kWidth #` switch must be used in conjunction with the `-o2` switch.
`-l` *`userkernel.dxe`*	The `-l` *`userkernel.dxe`* switch specifies the user boot kernel file. There is no default kernel for the ADSP-BF51x/BF52x/BF54x processors.
`-M`	The `-M` switch generates make dependencies only, no output file is generated.
`-maskaddr #`	The `-maskaddr #` switch masks all EPROM address bits above or equal to #. For example, `-maskaddr 29` (default) masks all the bits above and including `A29` (`AND`ed by `0x1FFF FFFF`). For example, `0x2000 0000` becomes `0x0000 0000`. The valid #s are integers `0` through `32`, but based on your specific input file, the value can be within a subset of [0, 32]. The `-maskaddr #` switch requires `-romsplitter` and affects the ROM section address only.
`-MaxBlockSize #`	The `-MaxBlockSize #` switch specifies the maximum block size up to `0x7FFFFFF0`. The value must be a multiple of 4. The default maximum block size is `0xFFF0` or the value specified by the `-MaxBlockSize` switch.
`-MaxFillBlockSize #`	The `-MaxFillBlockSize #` switch specifies the maximum fill block size up to `0xFFFFFF0`. The value must be a multiple of two. The default fill block size is `0xFFF0`.

Table 2-5. ADSP-BF51x/BF52x/BF54x Loader Command-Line Switch Summary (Cont'd)

Switch	Description
`-MM`	The `-MM` switch generates make dependencies while producing the output files.
`-Mo filename`	The `-Mo filename` switch writes make dependencies to the named file. Use the `-Mo` switch with either `-M` or `-MM`. If `-Mo` is absent, the default is a `<stdout>` display.
`-Mt target`	The `-Mt target` switch specifies the make dependencies target output file. Use the `-Mt` switch with either `-M` or `-MM`. If `-Mt` is not present, the default is the name of the input file with an `.ldr` extension.
`-NoInitCode`	The `-NoInitCode` switch directs the loader utility not to expect an init code file. The loader utility may expect an init code file, specified through the `-init filename.dxe` switch if the application has an external memory section. The init code file should contain the code to initialize registers for external memory initialization.
`-o filename`	The `-o filename` switch directs the loader utility to use the specified file as the name of the loader utility's output file. If the *filename* is absent, the default name is the root name of the input file with an `.ldr` extension.
`-o2`	The `-o2` switch directs the loader utility to produce two output files: one file for code from the initialization block and/or boot kernel and one file for user application code. To have a different format, boot mode, or output width for the application code output file, use the `-kb -kf -kwidth` switches to specify the boot mode, the boot format, and the boot width for the output kernel file, respectively. Combine `-o2` with `-l filename` and/or `-init filename.dxe`.
`-p #`	The `-p #` switch specifies a hex flash/PROM output start address for the application code. A valid value is between `0x0` and `0xFFFFFFFF`. A specified value must be greater than that specified by `-kp` if both kernel and/or initialization and application code are in the same output file (a single output file).

Table 2-5. ADSP-BF51x/BF52x/BF54x Loader Command-Line Switch Summary (Cont'd)

Switch	Description
`-proc` *`processor`*	The `-proc` *`processor`* switch specifies the target processor. The *`processor`* can be one of the processors listed in Table 2-1.
`-quickboot sec=`*`section`*	The `-quickboot` switch takes a *`sec=section`* (no spaces) assignment. The switch directs the loader utility to mark blocks within the LDF-defined output section name with the `BFLAG_QUICKBOOT` flag. The switch is used to mark blocks to skip on warm-boot cycles.
`-readall #`	The `-readall #` switch directs the loader utility to integrate fixed-position ROM sections within the loader boot stream. The switch calls the splitter utility as a transparent sub-process to the loader utility. Memory segments declared with the `TYPE(ROM)` command in the LDF file are processed by the splitter. Segments with the `TYPE(RAM)` command emit to the boot stream. The valid switch argument is an integer between `0` and `32`, where `29` is the default. In the resulting loader (`.ldr`) file in Intel hex-32 format, the ROM-based splitter data is merged with the RAM-based loader stream. The # argument is similar to the -maskaddr # switch, which designates the upper PROM address bit position for extended address mapping. The splitter utility is required to provide the -maskaddr # parameter to the loader utility to generate a ROM-based splitter stream, but the required splitter parameter is not available on the loader command line. The loader utility solves this requirement by supporting the `-readall #` switch.
`-romsplitter`	The `-romsplitter` switch creates a non-bootable image only. This switch overwrites the `-b` switch and any other switch bounded by the boot mode. In the `.ldf` file, declare memory segments to be 'split' as type `ROM`. The splitter skips `RAM` segments, resulting in an empty file if all segments are declared as `RAM`. The `-romsplitter` switch supports Intel hex and ASCII formats.

Table 2-5. ADSP-BF51x/BF52x/BF54x Loader Command-Line Switch Summary (Cont'd)

Switch	Description
`-save [sec=section]`	The `-save` switch takes a *`sec=section`* (no spaces) assignment. The switch directs the loader utility to mark blocks within the LDF-defined section name with the `BFLAG_SAVE` flag. The switch is used to mark blocks to archive for low-power or power-fail cycles.
`-si-revision #.#|none|any`	The `-si-revision {#.#|none|any}` switch provides a silicon revision of the specified processor. The switch parameter represents a silicon revision of the processor specified by the `-proc` *`processor`* switch. The parameter takes one of three forms: • The `#.#` value indicates one or more decimal digits, followed by a point, followed by one or two decimal digits. Examples of revisions are: `0.0`; `1.12`; `23.1`. Revision `0.0` is default. Revision `0.1` is distinct from and "lower" than revision `0.10`. The digits to the left of the point specify the chip tapeout number; the digits to the right of the point identify the metal mask revision number. The number to the right of the point cannot exceed decimal `255`. • The `none` value indicates that the VisualDSP++ ignores silicon errata. • The `any` value indicates that VisualDSP++ produces an output file that can be run at any silicon revision. The switch generates either a warning about any potential anomalous conditions or an error if any anomalous conditions occur. In the absence of the silicon revision switch, the loader utility selects the default silicon revision it is aware of, if any.
`-v`	The `-v` switch directs the loader utility to output verbose loader messages and status information as the loader processes files.
`-width #`	The `-width {8|16|32}` switch specifies an external memory device width (in bits) to the loader utility in flash/PROM boot mode (default is eight). For FIFO boot mode, the only valid width is `16`. For SPI, TWI, and UART boot modes, the only valid width is eight.

ADSP-BF51x/BF52x/BF54x Multi-DXE Loader Files

An ADSP-BF51x/BF52x/BF54x loader (`.ldr`) file can contain data of multiple application (`.dxe`) files. At boot time, the boot kernel boots one application file exclusively, or one application file initiates the boot of the next application file. In some cases, a single application can consist of multiple `.dxe` files.

Initialization code is a subroutine called at boot time. Unlike the ADSP-BF53x/BF56x processors, the ADSP-BF51x/BF52x/BF54x processors support initcode written in both assembly and C.

VisualDSP++ supports two methods of integrating multiple initcode subroutines:

- The -init filename.dxe command-line switch expects a `.dxe` file. The initcode is managed by a separate VisualDSP++ project. If the initcode is written in C language, ensure that the `.dxe` file does not include the CRT code because the boot kernel expects a subroutine.

  The -init filename.dxe switch can be used multiple times to specify the same file or different files a number of times. The loader utility places the code from the initialization files in the order the files appear on the command line. All initcodes are inserted after the first regular `.dxe` file.

  The loader utility equips every initcode with a dedicated first boot block, which has the `BFLAG_FIRST` flag set. Initcodes, however, do not feature a final block; they are terminated by a boot block,

tagged by the BFLAG_INIT flag. Therefore, in absence of the BFLAG_FINAL flag, the boot kernel continues processing of the subsequent .dxe data after finishing execution of the initcode.

- The -initcall sym=sym_symbol command-line switch relies on initcode subroutines that are part of the same VisualDSP++ project. Initcode subroutines invoked by the -initcall switch are not accompanied by any first boot blocks with the BFLAG_FIRST flag set. In the loader file, the initcode subroutines translate to boot blocks tagged by the BFLAG_INIT flag.

When writing an initcode subroutine in C, ensure that the code does not rely on libraries or heap support, which may not be available in memory by the time the initcode executes. An initcode routine is expected to return properly to the boot kernel by an RTS instruction and to meet C-language calling conventions (see the *VisualDSP++ 5.0 C/C++ Compiler and Library Manual for Blackfin Processors*).

Refer to the initcode examples provided with the VisualDSP++ installation in <*install_path*>\Blackfin\ldr\init_code.

Using VisualDSP++ Loader

After selecting a **Loader file (.ldr)** as the project output type for your Blackfin application on the **Application Settings** page in the VisualDSP++ **Project Wizard**, modify the default load settings.

The **Load** control in the **Project** tree control consists of multiple pages. When you open the **Load: Options** page (also called *loader property page*), view the default load settings for the selected processor. As an example, Figure 2-1 shows the ADSP-BF548 processor's default load settings for PROM boot mode. The dialog box options are equivalent to the command-line switches. Refer to "ADSP-BF51x/BF52x/BF54x Blackfin Loader Command-Line Switches" on page 2-8 for more information about the switches.

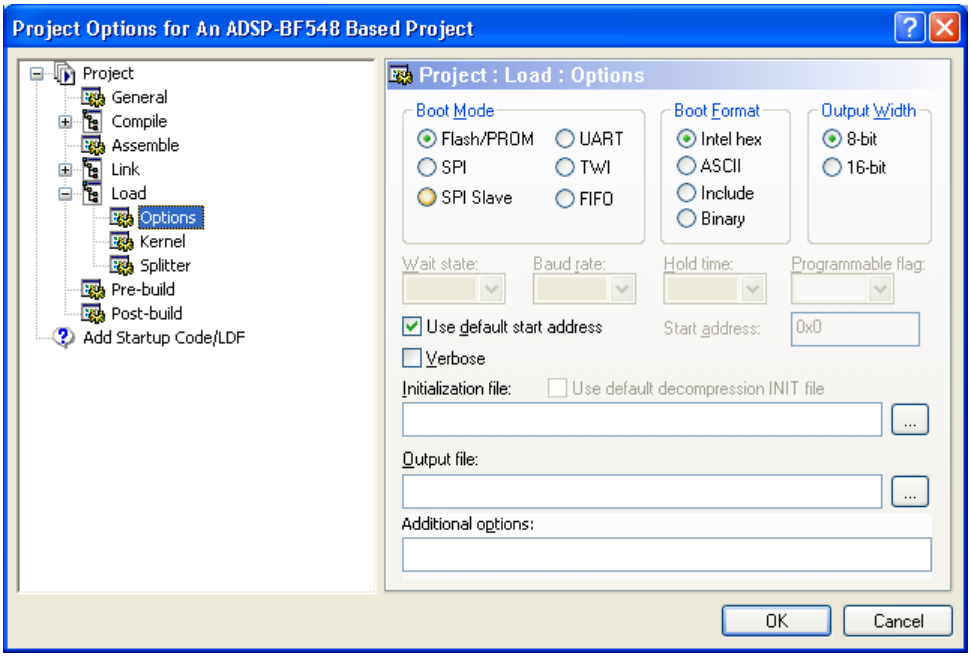

Figure 2-1. Project: Load: Options Page for ADSP-BF548 Processors

Using the page controls, select or modify the load settings. Table 2-6 describes each load control and corresponding setting. When satisfied with the settings, click **OK** to complete the load setup.

Table 2-6. Base Load Page Settings for ADSP-BF51x/BF52x/BF54x Processors

Setting	Description
Load	Selections for the loader utility. The selections are: • **Options** – default boot options (this section) • **Kernel** – specification for a second-stage loader (see page 2-20) • **Splitter** – specification for the no-boot mode (see page 2-22)
Boot mode	Specifies flash/PROM, SPI, SPI slave, UART, TWI, FIFO, SDRAM/DDR, OTP, or NAND as a boot source.

Table 2-6. Base Load Page Settings for ADSP-BF51x/BF52x/BF54x Processors (Cont'd)

Setting	Description
Boot format	Specifies Intel hex, ASCII, include, or binary format.
Output width	Specifies eight or 16 bits.
Use default start address	Uses the default flash/PROM output start address in hex format for the application code.
Start address	Specifies a flash/PROM output start address in hex format for the application code.
Verbose	Generates status information as the loader utility processes the files.
Initialization file	Directs the loader utility to include the initialization file (init code).
Output file	Names the loader utility's output file.
Additional options	Specifies additional loader switches. You can specify additional input files for a multi-input system. Type the input file names with the paths if the files are not in the current working directory, separate any two file names with a space in order for the loader utility to retrieve the files. **Note:** The loader utility processes the input files in the order in which the files appear on the command line, generated from the property page.

Using VisualDSP++ Second-Stage Loader

If you use a second-stage loader, select **Kernel** under **Load** in the **Project Options** tree control. The page shows the default settings for a loader file that does not include a second-stage loader.

Unless you develop a second-stage loader and use it, most of the options on the **Kernel** page are grayed out. Figure 2-2 shows a sample **Kernel** page with options set for an ADSP-BF548 Blackfin processor.

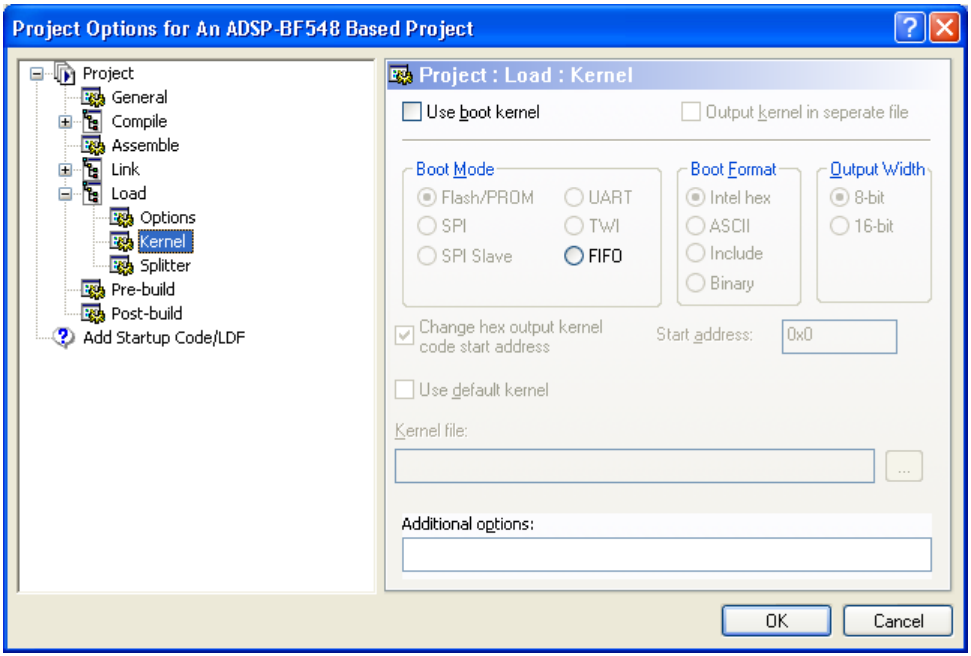

Figure 2-2. Project: Load: Kernel Page for ADSP-BF548 Processors

To create a loader file which includes a second-stage loader:

1. Select **Options** (under **Load**) to set up base load options (see "Using VisualDSP++ Loader" on page 2-18).
2. Select **Kernel** (under **Load**) to set up the second-stage loader options (Figure 2-2).
3. On the **Kernel** page, select **Use boot kernel**.
4. In **Kernel file**, enter the name of the second-stage loader file (`.dxe`).

5. To produce two output files, select the **Output kernel in separate file** check box. This option allows to boot the second-stage loader with an initialization code (if any) from one source and the application code from another source. You can specify the kernel output file options, such as the **Boot Mode** (source), **Boot Format**, and **Output Width**.
6. Select **Change hex output kernel code start address** to specify the **Start address** in hex format for the second-stage loader code. This option allows you to place the second-stage loader file at a specific location within the flash/PROM.
7. Click **OK** to complete the loader utility setup.

Using VisualDSP++ ROM Splitter

Unlike the loader utility, the splitter utility does not format the application data when transforming a `.dxe` file to an `.ldr` file. The splitter utility emits raw data only. Whether data and/or instruction sections are processed by the loader or by the splitter utility depends upon the LDF's `TYPE()` command. Sections declared with `TYPE(RAM)` are consumed by the loader utility, and sections declared by `TYPE(ROM)` are consumed by the splitter.

Figure 2-3 shows a sample **Load: Splitter** page with ROM splitter options. With the **Enable ROM splitter** box unchecked, only `TYPE(RAM)` segments are processed and all `TYPE(ROM)`) sections are ignored by the loader utility. If the box is checked, `TYPE(RAM)` sections are ignored, and `TYPE(ROM)` segments are processed by the splitter utility.

The **Mask Address** field masks all EPROM address bits above or equal to the number specified. For example, **Mask Address** = `29` (default) masks all bits above and including `A29` (`AND`ed by `0x1FFF FFFF`). Thus, `0x2000 0000` becomes `0x0000 0000`. The valid numbers are integers `0` through `32`; based on your specific input file, the value can be within a subset of [`0`, `32`].

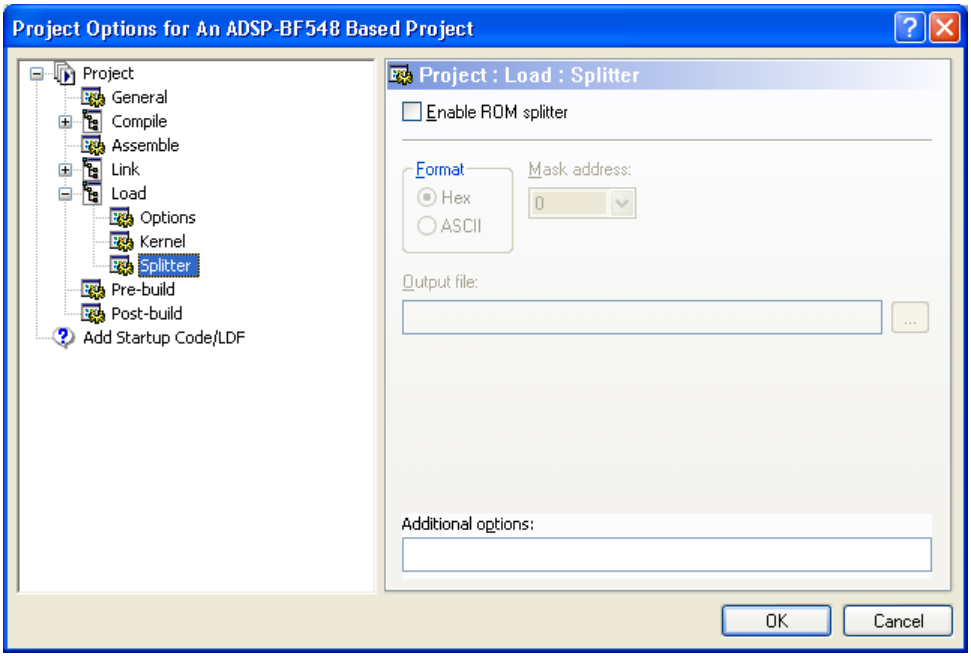

Figure 2-3. Project: Splitter Page for ADSP-BF548 Processors

3 LOADER/SPLITTER FOR ADSP-BF53X/BF561 BLACKFIN PROCESSORS

This chapter explains how the loader/splitter utility (`elfloader.exe`) is used to convert executable (`.dxe`) files into boot-loadable or non-bootable files for the ADSP-BF53x and ADSP-BF561 Blackfin processors.

Refer to "Introduction" on page 1-1 for the loader utility overview. Loader operations specific to the ADSP-BF53x and ADSP-BF561 Blackfin processors are detailed in the following sections.

- "ADSP-BF53x/BF561 Processor Booting" on page 3-2
  Provides general information on various boot modes, including information on the second-stage kernels.
- "ADSP-BF53x/BF561 Processor Loader Guide" on page 3-57
  Provides reference information on the loader utility's command-line syntax and switches.

ADSP-BF53x/BF561 Processor Booting

At power-up, after a reset, the processor transitions into a boot mode sequence configured by the BMODE pins. The BMODE pins are dedicated mode-control pins; that is, no other functions are performed by these pins. The pins can be read through bits in the system reset configuration register SYSCR.

An ADSP-BF53x or an ADSP-BF561 Blackfin processor can be booted from an eight- or 16-bit flash/PROM memory or from an eight-,16-, or 24-bit addressable SPI memory. The ADSP-BF561 processors does not support 24-bit addressable SPI memory boot. There is also a no-boot option (bypass mode) in which execution occurs from a 16-bit external memory.

- ADSP-BF531/BF532/BF533/BF534/BF536/BF537/ BF538/BF539 Processor Booting description is on page 3-3
- ADSP-BF535 Processor Booting description is on page 3-21
- ADSP-BF561 Processor Booting description is on page 3-35

Software developers who use the loader utility should be familiar with the the following operations.

- “ADSP-BF53x and ADSP-BF561 Multi-Application (Multi-DXE) Management” on page 3-46
- “ADSP-BF531/BF532/BF533/BF534/BF536/BF537 Processor Compression Support” on page 3-49

ADSP-BF531/BF532/BF533/BF534/BF536/BF537/ BF538/BF539 Processor Booting

Upon reset, an ADSP-BF531/BF532/BF533/BF534/BF536/BF537/ BF538/BF539 processor jumps to the on-chip boot ROM or jumps to 16-bit external memory for execution (if `BMODE = 0`) located at `0x2000 0000`. The ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processor On-Chip Boot ROM details can be found on page 3-23.

Table 3-1 summarizes the boot modes and execution start addresses for the ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF538, and ADSP-BF539 processors.

Table 3-2 summarizes the boot modes for the ADSP-BF534/BF536/BF537 processors, which in addition to all of the ADSP-BF531/BF532/BF533 processor boot modes, also can boot from a TWI serial device, a TWI host, and a UART host.

Table 3-1. Boot Mode Selections for ADSP-BF531/BF532/ BF533/BF538/BF539 Processors

Boot Source	BMODE[1:0]	Execution Start Address	
		ADSP-BF531 ADSP-BF532	ADSP-BF533 ADSP-BF538 ADSP-BF539
Executes from a 16-bit external ASYNC bank 0 memory (no-boot mode); see on page 3-16	00	0x2000 0000	0x2000 0000
Eight- or 16-bit flash/PROM	01	0xFFA0 8000	0xFFA0 0000
SPI host in SPI slave mode	10	0xFFA0 8000	0xFFA0 0000
Eight-, 16-, or 24-bit addressable SPI memory in SPI master boot mode with support for Atmel AT45DB041B, AT45DB081B, and AT45DB161B DataFlash devices	11	0xFFA0 8000	0xFFA0 0000

Table 3-2. ADSP-BF534/BF536/BF537 Processor Boot Modes

Boot Source	BMODE[2:0]
Executes from an external 16-bit memory connected to ASYNC bank 0; (no-boot mode or bypass on-chip boot ROM); see on page 3-16	000
Eight- or 16-bit flash/PROM	001
Reserved	010
Eight-, 16-, or 24-bit addressable SPI memory in SPI master mode with support for Atmel AT45DB041B, AT45DB081B, and AT45DB161B DataFlash devices	011
SPI host in SPI slave mode	100
TWI serial device	101
TWI host	110
UART host	111

- Execute from 16-bit external memory – execution starts from address 0x2000 0000 with 16-bit packing. The boot ROM is bypassed in this mode. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle R/W access times; 4-cycle setup).
- Boot from eight-bit or 16-bit external flash memory – the eight-bit or 16-bit flash boot routine located in boot ROM memory space is set up using asynchronous memory bank 0. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle R/W access times; 4-cycle setup). The boot ROM evaluates the first byte of the boot stream at address 0x2000 0000. If it is 0x40, eight-bit boot is performed. A 0x60 byte assumes a 16-bit memory device and performs eight-bit DMA. A 0x20 byte also assumes 16-bit memory but performs 16-bit DMA.
- Boot from serial SPI memory (EEPROM or flash) – eight-, 16-, or 24-bit addressable devices are supported as well as AT45DB041, AT45DB081, AT45DB161, AT45DB321, AT45DB642, and AT45DB1282 DataFlash® devices from Atmel. The SPI uses the PF10/SPI SSEL1 output pin to select a single SPI EEPROM/flash device, submits a read command and successive address bytes (0x00) until a valid eight-, 16-, or 24-bit, or Atmel addressable device is detected, and begins clocking data into the processor.
- Boot from SPI host device – the Blackfin processor operates in SPI slave mode and is configured to receive the bytes of the .ldr file from an SPI host (master) agent. To hold off the host device from transmitting while the boot ROM is busy, the Blackfin processor asserts a GPIO pin, called host wait (HWAIT), to signal the host device not to send any more bytes until the flag is deasserted. The flag is chosen by the user and this information is transferred to the Blackfin processor via bits 10:5 of the FLAG header.

- Boot from UART – using an autobaud handshake sequence, a boot-stream-formatted program is downloaded by the host. The host agent selects a baud rate within the UART's clocking capabilities. When performing the autobaud, the UART expects an "@" (boot stream) character (eight bits data, 1 start bit, 1 stop bit, no parity bit) on the `RXD` pin to determine the bit rate. It then replies with an acknowledgement that is composed of 4 bytes: `0xBF`, the value of `UART_DLL`, the value of `UART_DLH`, and `0x00`. The host can then download the boot stream. When the processor needs to hold off the host, it deasserts `CTS`. Therefore, the host must monitor this signal.
- Boot from serial TWI memory (EEPROM/flash) – the Blackfin processor operates in master mode and selects the TWI slave with the unique ID `0xA0`. It submits successive read commands to the memory device starting at two byte internal address `0x0000` and begins clocking data into the processor. The TWI memory device should comply with Philips I^2C Bus Specification version 2.1 and have the capability to auto-increment its internal address counter such that the contents of the memory device can be read sequentially.
- Boot from TWI host – the TWI host agent selects the slave with the unique ID `0x5F`. The processor replies with an acknowledgement, and the host can then download the boot stream. The TWI host agent should comply with Philips I^2C Bus Specification version 2.1. An I^2C multiplexer can be used to select one processor at a time when booting multiple processors from a single TWI.

To augment the boot modes, a secondary software loader can be added to provide additional booting mechanisms. The secondary loader could provide the capability to boot from flash, variable baud rate, and other sources.

The following loader topics also are discussed in this chapter.

- "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processor Boot Streams" on page 3-9
- "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processor Memory Ranges" on page 3-19

Refer to the processor's data sheet and hardware reference manual for more information on system configuration, peripherals, registers, and operating modes:

- Blackfin processor data sheets can be found at `http://www.analog.com/en/embedded-processing-dsp/blackfin/processors/data-sheets/resources/index.html`.
- Blackfin processor manuals can be found at `http://www.analog.com/en/embedded-processing-dsp/blackfin/processors/manuals/resources/index.html`.

ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processor On-Chip Boot ROM

The on-chip boot ROM for the ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 processors does the following.

1. Sets up supervisor mode by exiting the `RESET` interrupt service routine and jumping into the lowest priority interrupt (`IVG15`).

   Note that the on-chip boot ROM of the ADSP-BF534/BF536 and ADSP-BF537 processors executes at the Reset priority level, does not degrade to the lowest priority interrupt.

2. Checks whether the `RESET` was a software reset and, if so, whether to skip the entire sequence and jump to the start of L1 memory (`0xFFA0 0000` for the ADSP-BF533/BF534/BF536/BF537/BF538 and ADSP-BF539 processors; `0xFFA0 8000` for the

ADSP-BF531/BF532 processors) for execution. The on-chip boot ROM does this by checking the NOBOOT bit (bit 4) of the system reset configuration register (SYSCR). If bit 4 is not set, the on-chip boot ROM performs the full boot sequence. If bit 4 is set, the on-chip boot ROM bypasses the full boot sequence and jumps to the start of L1 memory.

3. The NOBOOT bit, if bit 4 of the SYSCR register is not set, performs the full boot sequence (Figure 3-1).

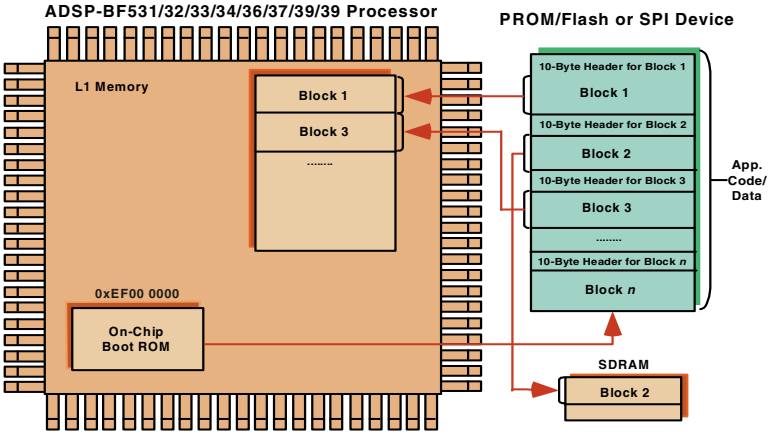

Figure 3-1. ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processors: Booting Sequence

The booting sequence for the ADSP-BF531/BF532/BF533/BF534/BF536/ BF537/BF538/BF539 processors is different from that for the ADSP-BF535 processors. The on-chip boot ROM for the former processors behaves similarly to the second-stage loader of the ADSP-BF535 processors (see on page 3-23). The boot ROM has the capability to parse address and count information for

each bootable block. This alleviates the need for a second-stage loader because a full application can be booted to the various memories with just the on-chip boot ROM.

The loader utility converts the application code (`.dxe`) into the loadable file by parsing the code and creating a file that consists of different blocks. Each block is encapsulated within a 10-byte header, which is illustrated in Figure 3-1 and detailed in the following section. The headers, in turn, are read and parsed by the on-chip boot ROM during booting.

The 10-byte header provides all information the on-chip boot ROM requires—where to boot the block to, how many bytes to boot in, and what to do with the block.

ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processor Boot Streams

The following sections describe the boot stream, header, and flag framework for the ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF534, ADSP-BF536, ADSP-BF537, ADSP-BF538, and ADSP-BF539 processors.

- "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Block Headers and Flags" on page 3-10
- "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Initialization Blocks" on page 3-13

The ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 processor boot stream is similar to the boot stream that uses a second-stage kernel of the ADSP-BF535 processors (detailed in "Loader Files With a Second-Stage Loader" on page 3-29). However, since the former processors do not employ a second-stage loader, their boot streams do not include the second-stage loader code and the associated 4-byte header on the top of the kernel code. There is also no 4-byte global header.

ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Block Headers and Flags

As the loader utility converts the code from an input `.dxe` file into blocks comprising the output loader file, each block receives a 10-byte header (Figure 3-2), followed by a block body (if a non-zero block) or no-block body (if a zero block). A description of the header structure can be found in Table 3-3.

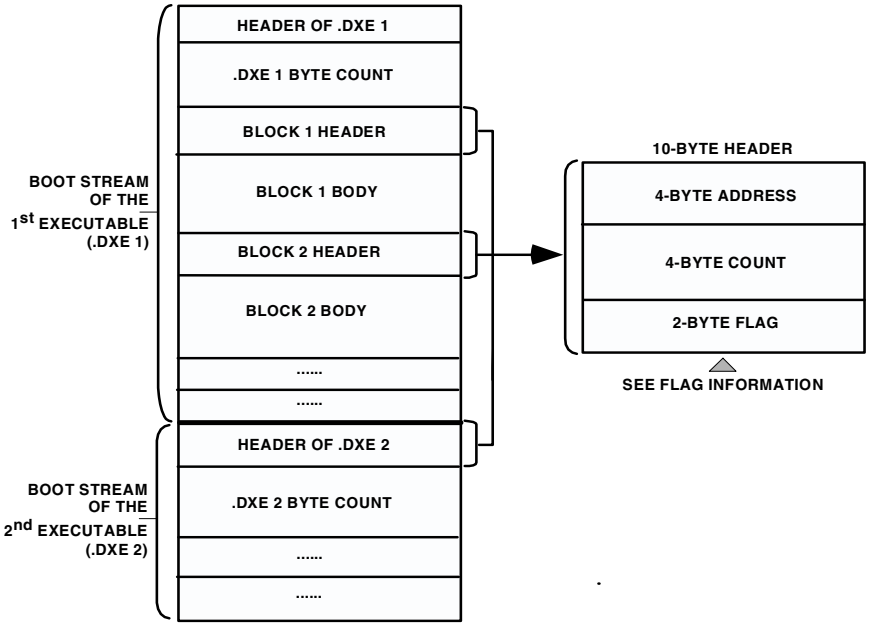

Figure 3-2. ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processors: Boot Stream Structure

Table 3-3. ADSP-BF531/BF532/BF533 Block Header Structure

Bit Field	Description
Address	4-byte address at which the block resides in memory

Table 3-3. ADSP-BF531/BF532/BF533 Block Header Structure (Cont'd)

Bit Field	Description
Count	4-byte number of bytes to boot
Flag	2-byte flag containing information about the block; the following text describes the flag structure

Refer to Figure 3-3 and Table 3-4 for the flag's bit descriptions.

Table 3-4. Flag Structure

Bit Field	Description
Zero-fill block	Indicates that the block is for a buffer filled with zeros. The body of a zero block is not included within the loader file. When the loader utility parses through the `.dxe` file and encounters a large buffer with zeros, it creates a zero-fill block to reduce the `.ldr` file size and boot time. If this bit is set, there is no block body in the block.
Processor type	Indicates the processor, either the ADSP-BF531/BF532/BF538 or the ADSP-BF533/BF534/BF536/BF537/BF539. Once booting is complete, the on-chip boot ROM jumps to `0xFFA0 0000` on the ADSP-BF533/BF536/BF537/BF538/BF539 processor and to `0xFFA0 8000` on the ADSP-BF531/BF532/ processors.
Initialization block	Indicates that the block is to be executed before booting. The initialization block indicator allows the on-chip boot ROM to execute a number of instructions before booting the actual application code. When the on-chip boot ROM detects an init block, it boots the block into internal memory and makes a `CALL` to it (initialization code must have an RTS at the end). This option allows the user to run initialization code (such as SDRAM initialization) before the full boot sequence proceeds. Figure 3-4 and Figure 3-5 illustrate the process. Initialization code can be included within the `.ldr` file by using the `-init` switch (see "-init filename.dxe" on page 3-62). See "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Initialization Blocks" on page 3-13 for more information.

Table 3-4. Flag Structure (Cont'd)

Bit Field	Description
Ignore block	Indicates that the block is not to be booted into memory; skips the block and moves on to the next one. Currently is not implemented for application code. This flag is equivalent to the FIRST flag in boot streams on the ADSP-BF51x/BF52x/BF54x processors. Because the IGNORE flag is used for other purposes on the ADSP-BF51x/BF52x/BF54x processors, the FIRST flag is invented to indicate the first header.
Compressed block	Indicates that the block contains compressed data. The compressed block can include a number of blocks compressed together to form a single compressed block.
Last block	Indicates that the block is the last block to be booted into memory. After the last block, the processor jumps to the start of L1 memory for application code execution. When it jumps to L1 memory for code execution, the processor is still in supervisor mode and in the lowest priority interrupt (IVG15).

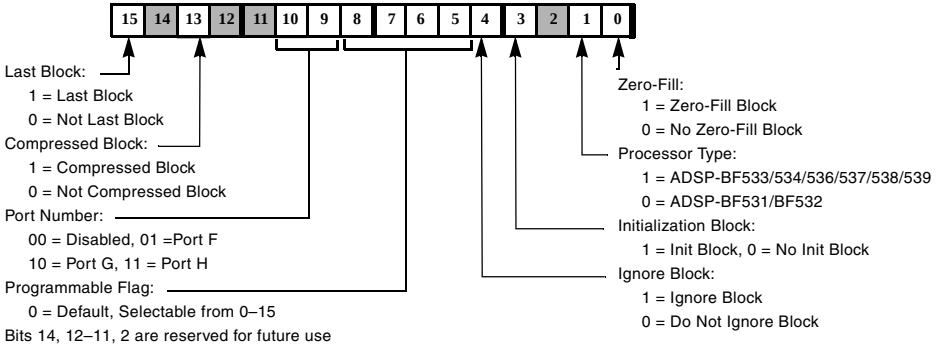
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Last Block:
1 = Last Block
0 = Not Last Block
Compressed Block:
1 = Compressed Block
0 = Not Compressed Block
Port Number:
00 = Disabled, 01 =Port F
10 = Port G, 11 = Port H
Programmable Flag:
0 = Default, Selectable from 0–15
Bits 14, 12–11, 2 are reserved for future use
Zero-Fill:
1 = Zero-Fill Block
0 = No Zero-Fill Block
Processor Type:
1 = ADSP-BF533/534/536/537/538/539
0 = ADSP-BF531/BF532
Initialization Block:
1 = Init Block, 0 = No Init Block
Ignore Block:
1 = Ignore Block
0 = Do Not Ignore Block

Figure 3-3. Flag Bit Assignments for 2-Byte Block Flag Word

Note that the ADSP-BF534/BF536/BF537 processor can have a special last block if the boot mode is two-wire interface (TWI). The loader utility saves all the data from 0xFF90 3F00 to 0xFF90 3FFF and makes the last block with the data. The loader utility, however, creates a regular last

block if no data is in that memory range. The space of `0xFF90 3F00` to `0xFF90 3FFF` is saved for the boot ROM to use as a data buffer during a boot process.

ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Initialization Blocks

The `-init` *`filename`* option directs the loader utility to produce the initialization blocks from the initialization section's code in the named file. The initialization blocks are placed at the top of a loader file. They are executed before the rest of the code in the loader file booted into the memory (see Figure 3-4).

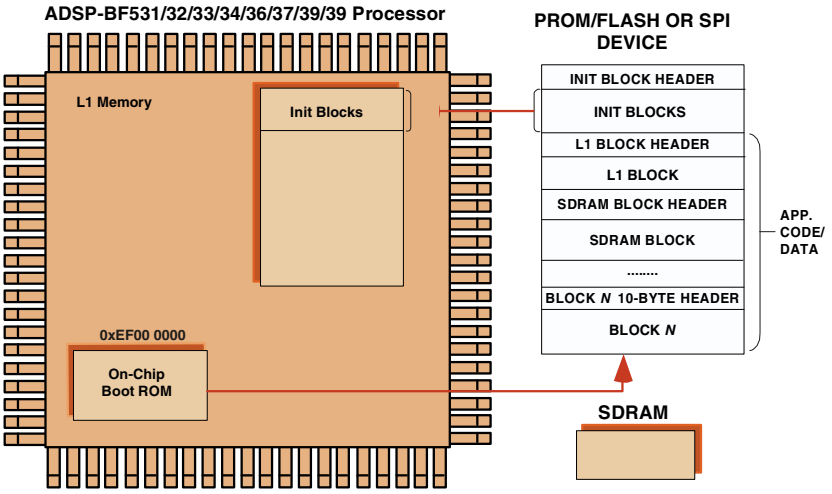

Figure 3-4. ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processors: Initialization Block Execution

Following execution of the initialization blocks, the boot process continues with the rest of data blocks until it encounters a final block (see Figure 3-5). The initialization code example follows in Listing 3-1 on page 3-14.

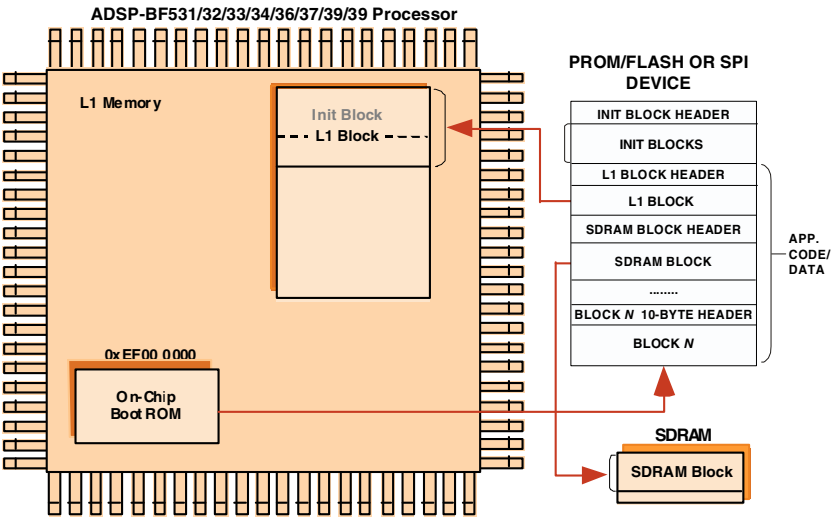

Figure 3-5. ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processors: Booting Application Code

Listing 3-1. Initialization Block Code Example

```
/*  This file contains 3 sections: */
/*  1) A Pre-Init Section-this section saves off all the
       processor registers onto the stack.
    2) An Init Code Section-this section is the initialization
       code which can be modified by the customer
       As an example, an SDRAM initialization code is supplied.
       The example setups the SDRAM controller as required by
       certain SDRAM types. Different SDRAMs may require
       different initialization procedure or values.
    3) A Post-Init Section-this section restores all the register
       from the stack. Customers should not modify the Pre-Init
       and Post-Init Sections. The Init Code Section can be
       modified for a particular application.*/
```

```
#include <defBF532.h>
.SECTION program;
/***********************Pre-Init Section************************/
[--SP] = ASTAT;   /* Stack Pointer (SP) is set to the end of */
[--SP] = RETS;    /* scratchpad memory (0xFFB00FFC) */
[--SP] = (r7:0);  /* by the on-chip boot ROM */
[--SP] = (p5:0);
[--SP] = I0;[--SP] = I1;[--SP] = I2;[--SP] = I3;
[--SP] = B0;[--SP] = B1;[--SP] = B2;[--SP] = B3;
[--SP] = M0;[--SP] = M1;[--SP] = M2;[--SP] = M3;
[--SP] = L0;[--SP] = L1;[--SP] = L2;[--SP] = L3;

/*******************Init Code Section*************************/
/*******Please insert Initialization code in this section******/
/***********************SDRAM Setup*****************************/
Setup_SDRAM:
     P0.L = LO(EBIU_SDRRC);
     /* SDRAM Refresh Rate Control Register */
     P0.H = HI(EBIU_SDRRC);
     R0 = 0x074A(Z);
     W[P0] = R0;
     SSYNC;

     P0.L = LO(EBIU_SDBCTL);
     /* SDRAM Memory Bank Control Register */
     P0.H = HI(EBIU_SDBCTL);
     R0 = 0x0001(Z);
     W[P0] = R0;
     SSYNC;

     P0.L = LO(EBIU_SDGCTL);
     /* SDRAM Memory Global Control Register */
     P0.H = HI(EBIU_SDGCTL);
     R0.L = 0x998D;
```

```
        R0.H = 0x0091;
        [P0] = R0;
        SSYNC;
/*********************Post-Init Section************************/
L3 = [SP++]; L2 = [SP++]; L1 = [SP++]; L0 = [SP++];
M3 = [SP++]; M2 = [SP++]; M1 = [SP++]; M0 = [SP++];
B3 = [SP++]; B2 = [SP++]; B1 = [SP++]; B0 = [SP++];
I3 = [SP++]; I2 = [SP++]; I1 = [SP++]; I0 = [SP++];
(p5:0) = [SP++];
(r7:0) = [SP++];
RETS = [SP++];
ASTAT = [SP++];
/*************************************************************/
RTS;
```

ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 and ADSP-BF535 Processor No-Boot Mode

The hardware settings of `BMODE = 00` for the ADSP-BF531, ADSP-BF532, and ADSP-BF533 processors or `BMODE = 000` for the ADSP-BF535 processors select the no-boot option. In this mode of operation, the on-chip boot kernel is bypassed after reset, and the processor starts fetching and executing instructions from address `0x2000 0000` in the asynchronous memory bank 0. The processor assumes 16-bit memory with valid instructions at that location.

To create a proper `.ldr` file that can be burned into either a parallel flash or EPROM device, you must modify the standard LDF file in order for the reset vector to be located accordingly. The following code fragments (Listing 3-6 and Listing 3-7) illustrate the required modifications in case of an ADSP-BF533 processor.

Listing 3-2. Section Assignment (LDF File) Example

```
MEMORY
{
  /* Off-chip Instruction ROM in Async Bank 0 */
  MEM_PROGRAM_ROM { TYPE(ROM) START(0x20000000) END(0x2009FFFF)
WIDTH(8) }
  /* Off-chip constant data in Async Bank 0  */
  MEM_DATA_ROM    { TYPE(ROM) START(0x200A0000) END(0x200FFFFF)
WIDTH(8) }
  /* On-chip SRAM data, is not booted automatically */
  MEM_DATA_RAM    { TYPE(RAM) START(0xFF903000) END(0xFF907FFF)
WIDTH(8) }
```

Listing 3-3. ROM Segment Definitions (LDF File) Example

```
PROCESSOR p0
{
  OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

  SECTIONS
  {
    program_rom
    {
        INPUT_SECTION_ALIGN(4)
        INPUT_SECTIONS( $OBJECTS(rom_code) )
    } >MEM_PROGRAM_ROM
    data_rom
    {
        INPUT_SECTION_ALIGN(4)
        INPUT_SECTIONS( $OBJECTS(rom_data) )
     } >MEM_DATA_ROM
    data_sram
    {
```

```
        INPUT_SECTION_ALIGN(4)
        INPUT_SECTIONS( $OBJECTS(ram_data) )
    } >MEM_DATA_RAM
```

With the LDF file modified this way, the source files can now take advantage of the newly-introduced sections, as in Listing 3-8.

Listing 3-4. Section Handling (Source File) Example

```
.SECTION rom_code;
_reset_vector: l0 = 0;
               l1 = 0;
               l2 = 0;
               l3 = 0;
               /* continue with setup and application code */
               /* . . . */
.SECTION rom_data;
.VAR myconst x = 0xdeadbeef;
               /* . . . */
.SECTION ram_data;
.VAR myvar y; /* note that y cannot be initialized automatically */
```

ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processor Memory Ranges

The on-chip boot ROM on the ADSP-BF531/BF532/BF533/BF534/BF536/ BF537/BF538/BF539 Blackfin processors allows booting to the following memory ranges.

- L1 memory
  - ADSP-BF531 processor:
    - Data bank A SRAM (0xFF80 4000–0xFF80 7FFF)
    - Instruction SRAM (0xFFA0 8000–0xFFA0 BFFF)
  - ADSP-BF532 processor:
    - Data bank A SRAM (0xFF80 4000–0xFF80 7FFF)
    - Data bank B SRAM (0xFF90 4000–0xFF90 7FFF)
    - Instruction SRAM (0xFFA0 8000–0xFFA1 3FFF)
  - ADSP-BF533 processor:
    - Data bank A SRAM (0xFF80 0000–0xFF80 7FFF)
    - Data bank B SRAM (0xFF90 000–0xFF90 7FFF)
    - Instruction SRAM (0xFFA0 0000–0xFFA1 3FFF)
  - ADSP-BF534 processor:
    - Data bank A SRAM (0xFF80 0000–0xFF80 7FFF)
    - Data bank B SRAM (0xFF90 0000–0xFF90 7FFF)
    - Instruction SRAM (0xFFA0 0000–0xFFA1 3FFF)
  - ADSP-BF536 processor:
    - Data bank A SRAM (0xFF80 4000–0xFF80 7FFF)
    - Data bank B SRAM (0xFF90 4000–0xFF90 7FFF)

- Instruction SRAM (0xFFA0 0000–0xFFA1 3FFF)

- ADSP-BF537 processor:
  - Data bank A SRAM (0xFF80 0000–0xFF80 7FFF)
  - Data bank B SRAM (0xFF90 0000–0xFF90 7FFF)
  - Instruction SRAM (0xFFA0 0000–0xFFA1 3FFF)
- ADSP-BF538 processor:
  - Data bank A SRAM (0xFF80 4000–0xFF80 7FFF)
  - Data bank B SRAM (0xFF90 4000–0xFF90 7FFF)
  - Instruction SRAM (0xFFA0 8000–0xFFA1 3FFF)
- ADSP-BF539 processor:
  - Data bank A SRAM (0xFF80 0000–0xFF80 3FFF)
  - Data bank B SRAM (0xFF90 2000–0xFF90 7FFF)
  - Instruction SRAM (0xFFA0 0000–0xFFA1 3FFF)

- SDRAM memory:
  - Bank 0 (0x0000 0000–0x07FF FFFF)

Booting to scratchpad memory (0xFFB0 0000) is not supported.

SDRAM must be initialized by user code before any instructions or data are loaded into it.

ADSP-BF535 Processor Booting

Upon reset, an ADSP-BF535 processor jumps to an external 16-bit memory for execution (if BMODE = 000) or to the on-chip boot ROM (if BMODE = 001, 010, or 011). The ADSP-BF535 Processor On-Chip Boot ROM details can be found on page 3-23. Table 3-5 summarizes the boot modes and code execution start addresses for the ADSP-BF535 processors.

Table 3-5. ADSP-BF535 Processor Boot Mode Selections

Boot Source	BMODE[2:0]	Execution Start Address
Executes from a 16-bit external memory (async bank 0); no-boot mode (bypass on-chip boot ROM); see on page 3-16.	000	0x2000 0000
Boots from an eight- or 16-bit flash memory	001	0xF000 0000[1]
Boots from an eight-bit address SPI0 serial EEPROM	010	0xF000 0000[1]
Boots from a 16-bit address SPI0 serial EEPROM	011	0xF000 0000[1]
Reserved	111–100	N/A

1 The processor jumps to this location after the booting is complete.

- Execute from 16-bit external memory—execution starts from address 0x2000000 with 16-bit packing. The boot ROM is bypassed in this mode.
- Boot from eight-bit external flash memory—the eight-bit flash boot routine located in boot ROM memory space is set up using asynchronous memory bank 0. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle R/W access times; 4-cycle setup).

- Boot from SPI serial EEPROM (eight-bit addressable)—the `SPI0` uses `PF10` output pin to select a single SPI EPROM device, submits a read command at address `0x00`, and begins clocking data into the beginning of L2 memory. An eight-bit addressable SPI compatible EPROM must be used.
- Boot from SPI serial EEPROM (16-bit addressable)— the `SPI0` uses `PF10` output pin to select a single SPI EPROM device, submits a read command at address `0x0000`, and begins clocking data into the beginning of L2 memory. A 16-bit addressable SPI compatible EPROM must be used.

A secondary software loader is provided to augment the boot modes. The secondary loader provides the capability to boot from PCI, 16-bit flash memory, fast flash, variable baud rates, and so on. The ADSP-BF535 Processor Second-Stage Loader is detailed on page 3-24.

The following ADSP-BF535 topics also are discussed in this chapter.

- "ADSP-BF535 Processor Boot Streams" on page 3-27
- "ADSP-BF535 Processor Memory Ranges" on page 3-33

Refer to the *ADSP-BF535 Blackfin Embedded Processor* data sheet and the *ADSP-BF535 Blackfin Processor Hardware Reference* manual for more information on system configuration, peripherals, registers, and operating modes.

ADSP-BF535 Processor On-Chip Boot ROM

The on-chip boot ROM for the ADSP-BF535 processor does the following (Figure 3-6).

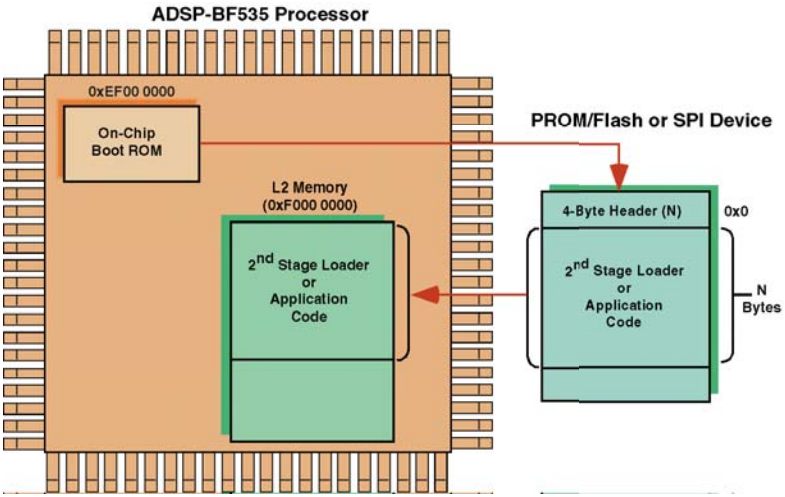

Figure 3-6. ADSP-BF535 Processors: On-Chip Boot ROM

1. Sets up supervisor mode by exiting the `RESET` interrupt service routine and jumping into the lowest priority interrupt (`IVG15`).
2. Checks whether the `RESET` is a software reset and if so, whether to skip the entire boot sequence and jump to the start of L2 memory (`0xF000 0000`) for execution. The on-chip boot ROM does this by checking bit 4 of the system reset configuration register (`SYSCR`). If bit 4 is not set, the on-chip boot ROM performs the full boot sequence. If bit 4 is set, the on-chip boot ROM bypasses the full boot sequence and jumps to `0xF000 0000`.

3. Finally, if bit 4 of the SYSCR register is not set, performs the full boot sequence. The full boot sequence includes:
   - Checking the boot source (either flash/PROM or SPI memory) by reading BMODE2-0 from the SYSCR register.
   - Reading the first four bytes from location 0x0 of the external memory device. These four bytes contain the byte count (*N*), which specifies the number of bytes to boot in.
   - Booting in *N* bytes into internal L2 memory starting at location 0xF000 0000.
   - Jumping to the start of L2 memory for execution.

The on-chip boot ROM boots in *N* bytes from the external memory. These *N* bytes can define the size of the actual application code or a second-stage loader that boots in the application code.

ADSP-BF535 Processor Second-Stage Loader

The only situation where a second-stage loader is unnecessary is when the application code contains only one section starting at the beginning of L2 memory (0xF000 0000).

A second-stage loader must be used in applications in which multiple segments reside in L2 memory or in L1 memory and/or SDRAM. In addition, a second-stage loader must be used to change the wait states or hold time cycles for a flash/PROM booting or to change the baud rate for an SPI boot (see "Blackfin Loader Command-Line Switches" on page 3-60 for more information on these features). Some Second-Stage Loader Restrictions are documented on page 3-34.

When a second-stage loader is used for booting, the following sequence occurs.

1. Upon reset, the on-chip boot ROM downloads `N` bytes (the second-stage loader) from external memory to address `0xF000 0000` in L2 memory (Figure 3-7).

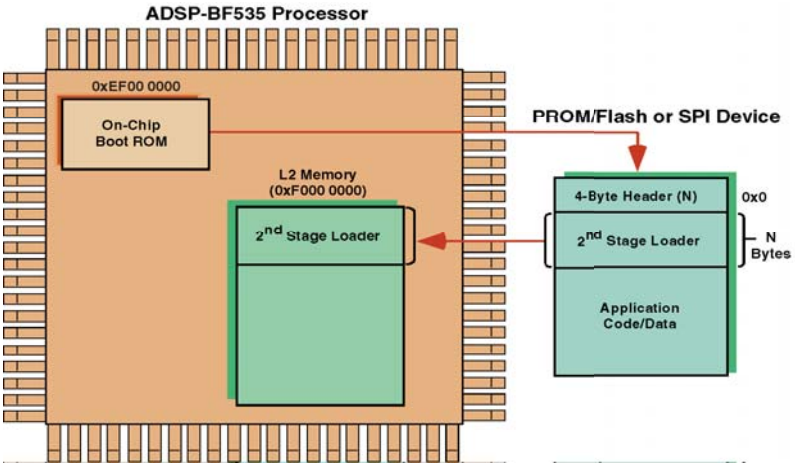

Figure 3-7. ADSP-BF535 Processors: Booting With Second-Stage Loader

2. The second-stage loader copies itself to the bottom of L2 memory.

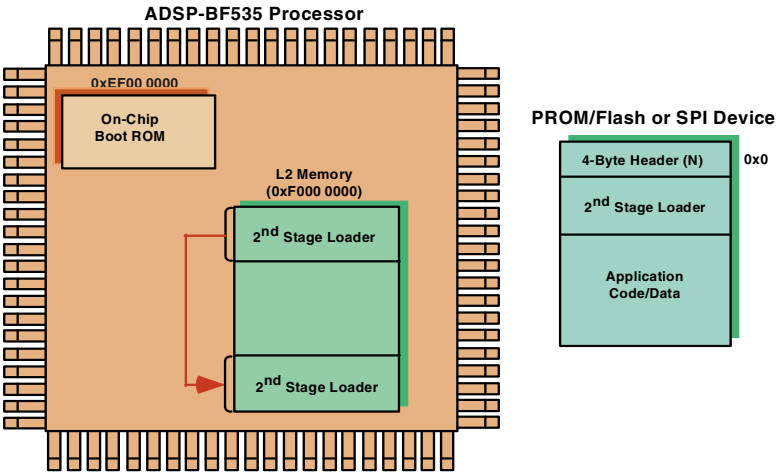

Figure 3-8. ADSP-BF535 Processors: Copying Second-Stage Loader

3. The second-stage loader downloads the application code and data into the various memories of the Blackfin processor (Figure 3-9).

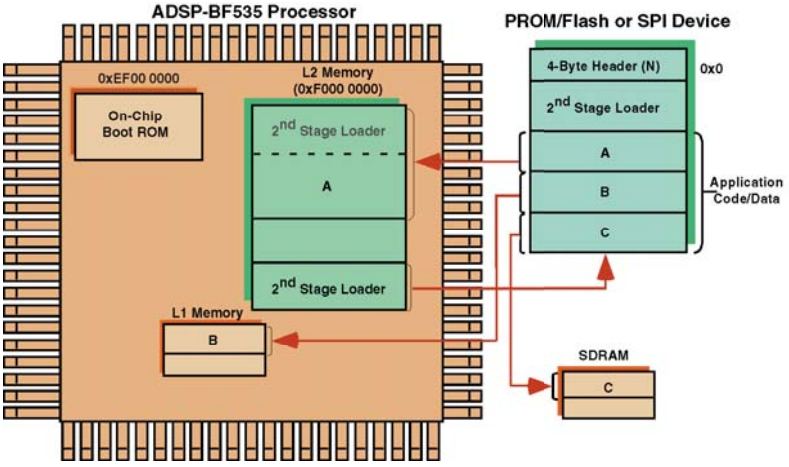

Figure 3-9. ADSP-BF535 Processors: Booting Application Code

4. Finally, after booting, the second-stage loader jumps to the start of L2 memory (`0xF000 0000`) for application code execution (Figure 3-10).

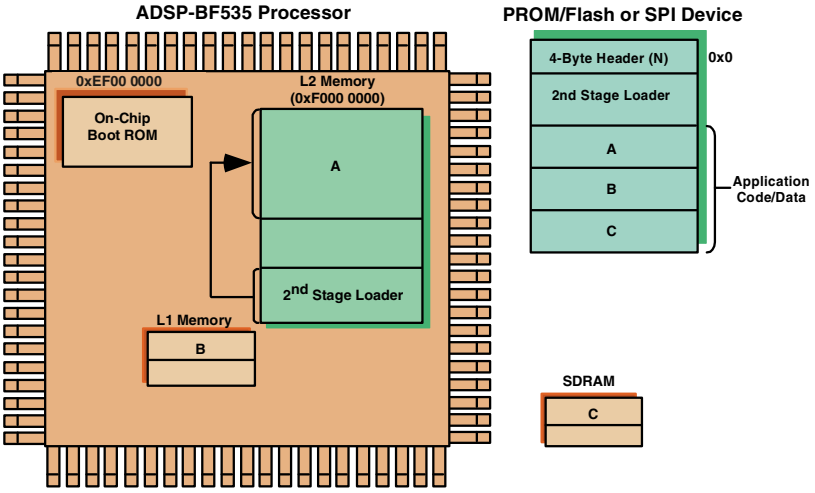

Figure 3-10. ADSP-BF535 Processors: Starting Application Code

ADSP-BF535 Processor Boot Streams

The loader utility generates the boot stream and places the boot stream in the output loader (`.ldr`) file. The loader utility prepares the boot stream in a way that enables the on-chip boot ROM and the second-stage loader to load the application code and data to the processor memory correctly. Therefore, the boot stream contains not only user application code but also header and flag information that is used by the on-chip boot ROM and the second-stage loader.

The following diagrams illustrate boot streams utilized by the ADSP-BF535 processor's boot kernel:

- "Loader Files Without a Second-Stage Loader" on page 3-28
- "Loader Files With a Second-Stage Loader" on page 3-29
- "Global Headers" on page 3-31
- "Block Headers and Flags" on page 3-32

Loader Files Without a Second-Stage Loader

Figure 3-11 is a graphical representation of an output loader file for eight-bit flash/PROM boot and eight-/16-bit addressable SPI boot without the second-stage loader.

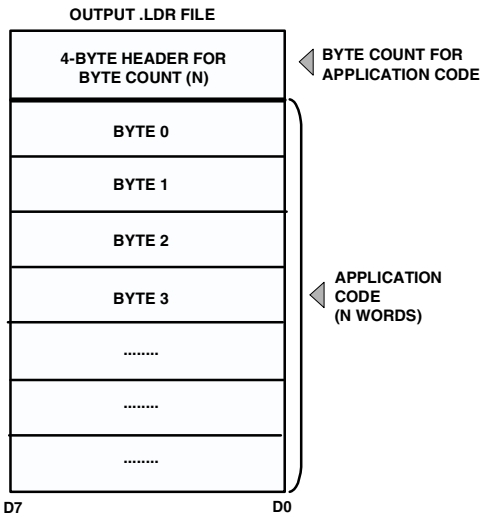

Figure 3-11. Loader File for Eight-bit Flash/PROM and SPI Boot Without Second-Stage Loader

Figure 3-12 is a graphical representation of an output loader file for 16-bit flash/PROM boot without the second-stage loader.

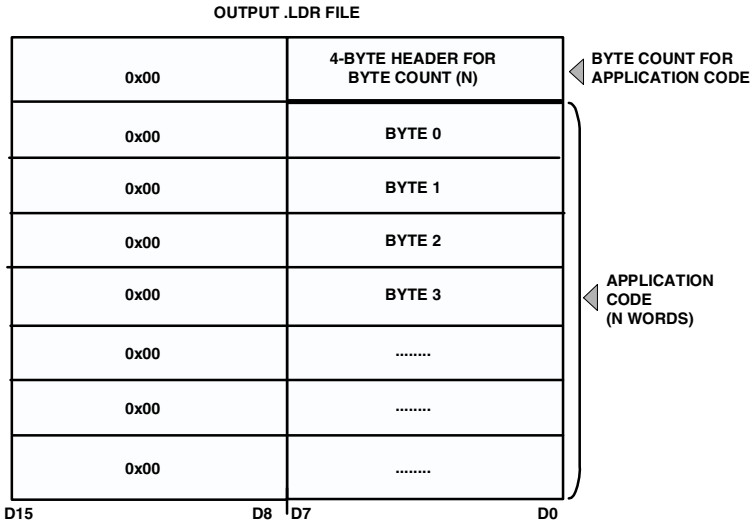

Figure 3-12. Loader File for 16-bit Flash/PROM Boot Without Second-Stage Loader

Loader Files With a Second-Stage Loader

Figure 3-13 is a graphical representation of an output loader file for eight-bit flash/PROM boot and eight- or 16-bit addressable SPI boot with the second-stage loader.

Figure 3-14 is graphical representation of an output loader file for 16-bit flash/PROM boot with the second-stage loader.

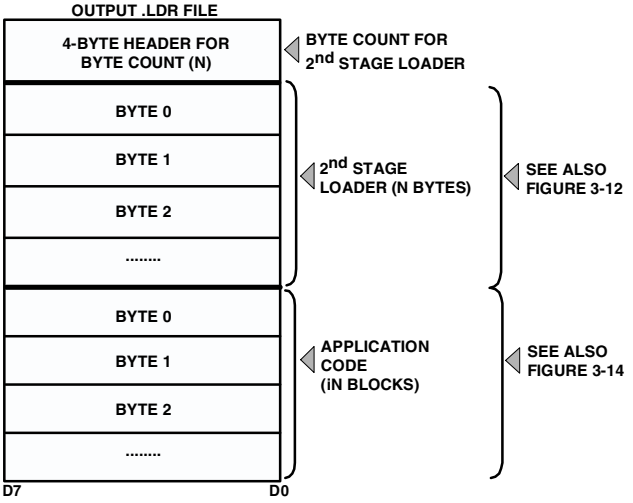

Figure 3-13. Loader File for Eight-bit Flash/PROM and SPI Boot With Second-Stage Loader

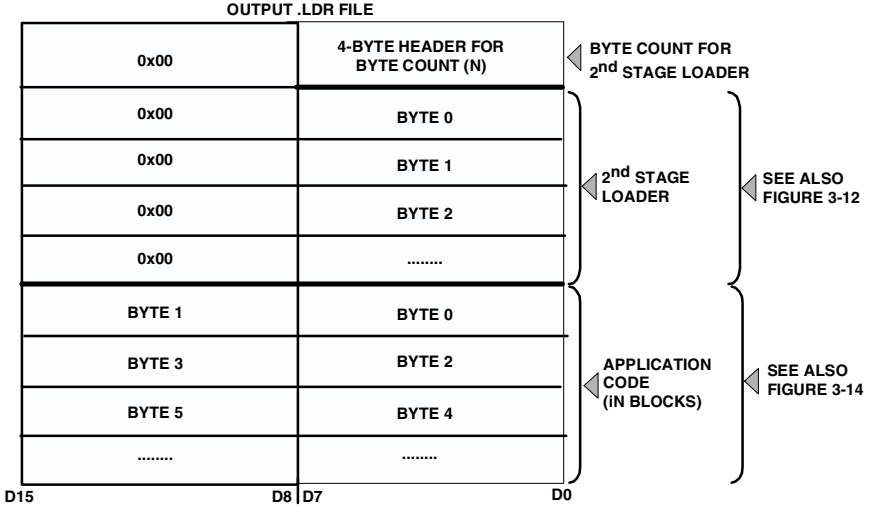

Figure 3-14. Loader File for 16-bit Flash/PROM Boot With Second-Stage Loader

Global Headers

Following the second-stage loader code and address in a loader file, there is a 4-byte global header. The header provides the global settings for a booting process (see Figure 3-15).

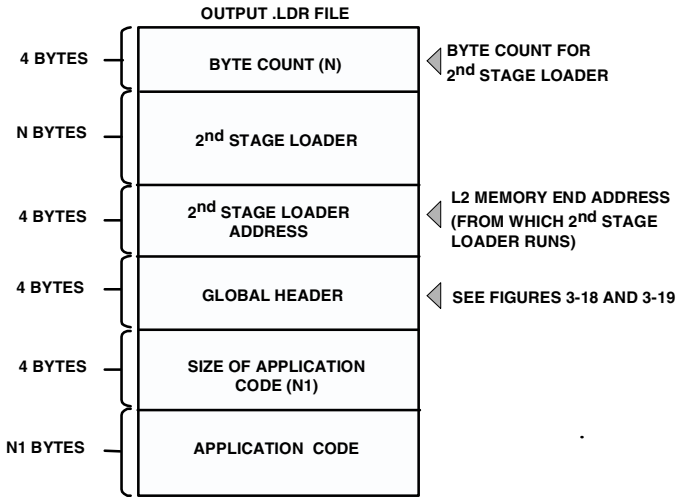

Figure 3-15. Global Header

A global header's bit assignments for eight- and 16-bit Flash/PROM boot are illustrated in Figure 3-16.

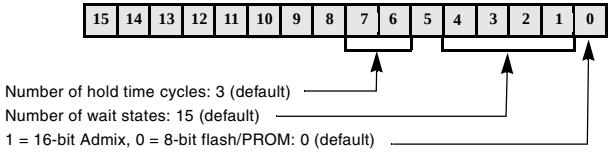
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Number of hold time cycles: 3 (default)
Number of wait states: 15 (default)
1 = 16-bit Admix, 0 = 8-bit flash/PROM: 0 (default)

Figure 3-16. Flash/PROM Boot: Global Header Bit Assignments

A global header's bit assignments for eight- and 16-bit addressable SPI boot are illustrated in Figure 3-17.

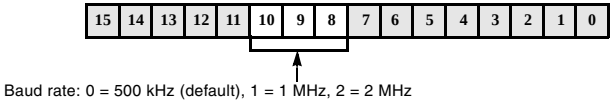

Figure 3-17. SPI Boot: Global Header Bit Assignments

Block Headers and Flags

For application code, a block is the basic structure of the output `.ldr` file when the second-stage loader is used. All application code is grouped into blocks. A block always has a header and a body if it is a non-zero block. A block does not have a body if it is a zero block. A block structure is illustrated in Figure 3-18.

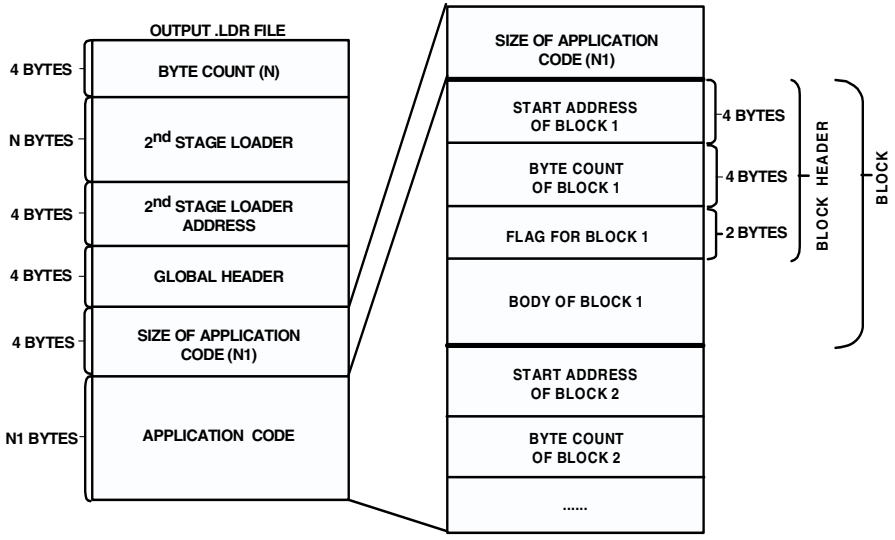

Figure 3-18. An Application Block

A block header has three words: 4-byte clock start address, 4-byte block byte count, and 2-byte flag word.

The ADSP-BF535 block flag word's bits are illustrated in Figure 3-19.

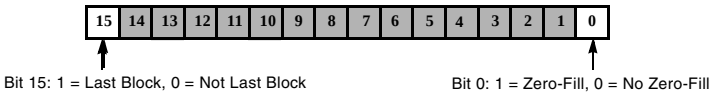

Figure 3-19. Block Flag Word Bit Assignments

ADSP-BF535 Processor Memory Ranges

Second-stage loaders are available for the ADSP-BF535 processors in VisualDSP++ 3.0 and higher. They allow booting to:

- L2 memory (0xF000 0000)
- L1 memory
  - Data bank A SRAM (0xFF80 0000)
  - Data bank B SRAM (0xFF90 0000)
  - Instruction SRAM (0xFFA0 0000)
  - Scratchpad SRAM (0xFFB0 0000)
- SDRAM
  - Bank 0 (0x0000 0000)
  - Bank 1 (0x0800 0000)
  - Bank 2 (0x1000 0000)
  - Bank 3 (0x1800 0000)

SDRAM must be initialized by user code before any instructions or data are loaded into it.

Second-Stage Loader Restrictions

Using the second-stage loader imposes the following restrictions.

- The bottom of L2 memory must be reserved during booting. These locations can be reallocated during runtime. The following locations pertain to the current second-stage loaders.
  - For eight- and 16-bit flash/PROM booting, reserve `0xF003 FE00–0xF003 FFFF` (last 512 bytes).
  - For eight- and 16-bit addressable SPI booting, reserve `0xF003 FD00–0xF003 FFFF` (last 768 bytes).
- If segments reside in SDRAM memory, configure the SDRAM registers accordingly in the second-stage loader before booting.
  - Modify a section of code called "`SDRAM setup`" in the second-stage loader and rebuild the second-stage loader.
- Any segments residing in L1 instruction memory (`0xFFA0 0000-0xFFA0 3FFF`) must be eight-byte aligned.
  - Declare segments, within the `.ldf` file, that reside in L1 instruction memory at starting locations that are eight-byte aligned (for example, `0xFFA0 0000`, `0xFFA0 0008`, `0xFFA0 0010`, and so on).
  - Use the `.ALIGN 8;` directives in the application code.

The two reasons for these restrictions are:

- Core writes into L1 instruction memory are not allowed.
- DMA from an eight-bit external memory is not possible since the minimum width of the external bus interface unit (EBIU) is 16 bits.

Load bytes into L1 instruction memory by using the instruction test command and data registers, as described in the Memory chapter of the appropriate hardware reference manual. These registers transfer eight-byte sections of data from external memory to internal L1 instruction memory.

ADSP-BF561 Processor Booting

The booting sequence for the ADSP-BF561 dual-core processors is similar to the ADSP-BF531/BF532/BF533 processor boot sequence described on page 3-7. Differences occur because the ADSP-BF561 processor has two cores: core A and core B. After reset, core B remains idle, but core A executes the on-chip boot ROM located at address `0xEF00 0000`.
The ADSP-BF561 Processor On-Chip Boot ROM details can be found on page 3-23.

Table 3-6 summarizes the boot modes and execution start addresses for the ADSP-BF561 processors.

Table 3-6. ADSP-BF561 Processor Boot Mode Selections

Boot Source	BMODE[1:0]
16-bit external memory (bypass boot ROM)	00
Eight- or 16-bit flash	01
SPI host	10
SPI serial EEPROM (16-bit address range)	11

- Execute from 16-bit external memory – execution starts from address `0x2000 0000` with 16-bit packing. The boot ROM is bypassed in this mode. All configuration settings are set for the slowest device possible (3-cycle hold time, 15-cycle R/W access times, 4-cycle setup).

- Boot from eight-bit/16-bit external flash memory – the eight-bit/16-bit flash boot routine located in boot ROM memory space is set up using asynchronous memory bank 0. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle R/W access times; 4-cycle setup).
- Boot from SPI host – the ADSP-BF561 processor is configured as an SPI slave device and a host is used to boot the processor. The host drives the SPI clock and is therefore responsible for the timing. The baud rate should be equal to or less than one fourth of the ADSP-BF561 system clock (`SCLK`).
- Boot from SPI serial EEPROM (16-bit addressable) – the SPI uses the `PF2` output pin to select a single SPI EPROM device, submits a read command at address `0x0000`, and begins clocking data into the beginning of L1 instruction memory. A 16-bit/24-bit addressable SPI-compatible EPROM must be used.

The following loader topics also are discussed in this chapter.

- "ADSP-BF561 Processor Boot Streams" on page 3-37
- "ADSP-BF561 Processor Initialization Blocks" on page 3-43
- "ADSP-BF561 Dual-Core Application Management" on page 3-44
- "ADSP-BF561 Processor Memory Ranges" on page 3-45

Refer to the *ADSP-BF561 Embedded Symmetric Multiprocessor* data sheet and the *ADSP-BF561 Blackfin Processor Hardware Reference* manual for information about the processor's operating modes and states, including background information on system reset and booting.

ADSP-BF561 Processor On-Chip Boot ROM

The boot ROM loads an application program from an external memory device and starts executing that program by jumping to the start of core A's L1 instruction SRAM, at address `0xFFA0 0000`.

Similar to the ADSP-BF531/BF532/BF533 processor, the ADSP-BF561 boot ROM uses the interrupt vectors to stay in supervisor mode.

The boot ROM code transitions from the `RESET` interrupt service routine into the lowest priority user interrupt service routine (`Int 15`) and remains in the interrupt service routine. The boot ROM then checks whether it has been invoked by a software reset by examining bit 4 of the system reset configuration register (`SYSCR`).

If bit 4 is not set, the boot ROM presumes that a hard reset has occurred and performs the full boot sequence. If bit 4 is set, the boot ROM understands that the user code has invoked a software reset and restarts the user program by jumping to the beginning of core A's L1 memory (`0xFFA0 0000`), bypassing the entire boot sequence.

When developing an ADSP-BF561 processor application, you start with compiling and linking your application code into an executable (`.dxe`) file. The debugger loads the `.dxe` file into the processor's memory and executes it. With two cores, two `.dxe` files can be loaded at once. In the real-time environment, there is no debugger which allows the boot ROM to load the executables into memory.

ADSP-BF561 Processor Boot Streams

The loader utility converts the `.dxe` file into a boot stream (`.ldr`) file by parsing the executable and creating blocks. Each block is encapsulated within a 10-byte header. The `.ldr` file is burned into the external memory device (flash memory, PROM, or EEPROM). The boot ROM reads the external memory device, parsing the headers and copying the blocks to the

addresses where they reside during program execution. After all the blocks are loaded, the boot ROM jumps to address 0xFFA0 0000 to execute the core A program.

When code is run on both cores, the core A program is responsible for releasing core B from the idle state by clearing bit 5 in core A's system configuration register. Then core B begins execution at address 0xFF60 0000.

Multiple .dxe files are often combined into a single boot stream (see "ADSP-BF561 Dual-Core Application Management" on page 3-44 and "ADSP-BF53x and ADSP-BF561 Multi-Application (Multi-DXE) Management" on page 3-46).

Unlike the ADSP-BF531/BF532/BF533 processor, the ADSP-BF561 boot stream begins with a 4-byte global header, which contains information about the external memory device. A bit-by-bit description of the global header is presented in Table 3-7. The global header also contains a signature in the upper 4 bits that prevents the boot ROM from reading in a boot stream from a blank device.

Table 3-7. ADSP-BF561 Global Header Structure

Bit Field	Description
0	1 = 16-bit flash, 0 = eight-bit flash; default is 0
1-4	Number of wait states; default is 15
5	Unused bit
6-7	Number of hold time cycles for flash; default is 3
8-10	Baud rate for SPI boot: 00 = 500k, 01 = 1M, 10 = 2M
11-27	Reserved for future use
28-31	Signature that indicates valid boot stream

Following the global header is a `.dxe` count block, which contains a 32-bit byte count for the first `.dxe` file in the boot stream. Though this block contains only a byte count, it is encapsulated by a 10-byte block header, just like the other blocks.

The 10-byte header instructs the boot ROM where, in memory, to place each block, how many bytes to copy, and whether the block needs any special processing. The block header structure is the same as that of the ADSP-BF531/BF532/BF533 processors (described in "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Block Headers and Flags" on page 3-10). Each header contains a 4-byte start address for the data block, a 4-byte count for the data block, and a 2-byte flag word, indicating whether the data block is a "zero-fill" block or a "final block" (the last block in the boot stream).

For the `.dxe` count block, the address field is irrelevant since the block is not going to be copied to memory. The "ignore bit" is set in the flag word of this header, so the boot loader utility does not try to load the `.dxe` count but skips the count. For more details, see "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Block Headers and Flags" on page 3-10.

Following the `.dxe` count block are the rest of the blocks of the first `.dxe`.

A bit-by-bit description of the boot steam is presented in Table 3-8. When learning about the ADSP-BF561 boot stream structure, keep in mind that the count byte for each `.dxe` is, itself, a block encapsulated by a block header.

Table 3-8. ADSP-BF561 Processor Boot Stream Structure

Bit Field	Description	
0-7	LSB of the global header	32-Bit Global Header
8-15	8–15 of the global header	
16-23	16–23 of the global header	
24-31	MSB of the global header	
32-39	LSB of the address field of 1st .dxe count block (no care)	10-Byte .dxe1 Header
40-47	8–15 of the address field of 1st .dxe count block (no care)	
48-55	16–23 of the address field of 1st .dxe count block (no care)	
56-63	MSB of the address field of 1st .dxe count block (no care)	
64-71	LSB (4) of the byte count field of 1st .dxe count block	
72-79	8–15 (0) of the byte count field of 1st .dxe count block	
80-87	16–23 (0) of the byte count field of 1st .dxe count block	
88-95	MSB (0) of the byte count field of 1st .dxe count block	
96-103	LSB of the flag word of 1st .dxe count block – ignore bit set	
104-111	MSB of the flag word of 1st .dxe count block	
112-119	LSB of the first 1st .dxe byte count	32-Bit Block Byte Count
120-127	8–15 of the first 1st .dxe byte count	
128-135	16–23 of the first 1st .dxe byte count	
136-143	24–31 of the first 1st .dxe byte count	

1-0-Byte Block Header	144-151	LSB of the address field of the 1st data block in 1st .dxe	.dxe1 Block Data
	152-159	8–15 of the address field of the 1st data block in 1st .dxe	
	160-167	16–23 of the address field of the 1st data block in 1st .dxe	
	168-175	MSB of the address field of the 1st data block in 1st .dxe	
	176-183	LSB of the byte count of the 1st data block in 1st .dxe	
	184-191	8–15 of the byte count of the 1st data block in 1st .dxe	
	192-199	16–23 of the byte count of the 1st data block in 1st .dxe	
	200-207	MSB of the byte count of the 1st data block in 1st .dxe	
	208-215	LSB of the flag word of the 1st block in 1st .dxe	
	216-223	MSB of the flag word of the 1st block in 1st .dxe	

Block Data	224-231	Byte 3 of the 1st block of 1st .dxe	.dxe1 Block Data (Cont'd)
	232-239	Byte 2 of the 1st block of 1st .dxe	
	240-247	Byte 1 of the 1st block of 1st .dxe	
	248-255	Byte 0 of the 1st block of 1st .dxe	
	256-263	Byte 7 of the 1st block of 1st .dxe	
	…	And so on …	

10-Byte Block Header	…	LSB of the address field of the nth data block in 1st `.dxe`	.dxe1 Block Data (Cont'd)
	…	8–15 of the address field of the nth data block in 1st `.dxe`	
	…	16–23 of the address field of the nth data block in 1st `.dxe`	
	…	MSB of the address field of the nth data block in 1st `.dxe`	
	…	LSB of the byte count of the nth data block in 1st `.dxe`	
	…	8–15 of the byte count of the nth data block in 1st `.dxe`	
	…	16–23 of the byte count of the nth data block in 1st `.dxe`	
	…	MSB of the byte count of the nth data block in 1st `.dxe`	
	…	LSB of the flag word of the nth block in 1st `.dxe`	
	…	MSB of the flag word of the nth block in 1st `.dxe`	

Block Data	…	And so on …	.dxe1 Block Data (Cont'd)
	…	Byte 1 of the nth block of 1st `.dxe`	
	…	Byte 0 of the nth block of 1st `.dxe`	

…	LSB of the address field of 2nd `.dxe` count block (no care)	10-Byte .dxe2 Header
…	8–15 of the address field of 2nd `.dxe` count block (no care)	
…	And so on…	

ADSP-BF561 Processor Initialization Blocks

The initialization block or a second-stage loader utility must be used to initialize the SDRAM memory of the ADSP-BF561 processor before any instructions or data are loaded into it.

The initialization blocks are identified by a bit in the flag word of the 10-byte block header. When the boot ROM encounters the initialization blocks in the boot stream, it loads the blocks and executes them immediately. The initialization blocks must save and restore registers and return to the boot ROM, so the boot ROM can load the rest of the blocks. For more details, see "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Block Headers and Flags" on page 3-10.

Both the initialization block and second-stage loader utility can be used to force the boot ROM to load a specific `.dxe` file from the external memory device if the boot ROM stores multiple executable files. The initialization block can manipulate the `R0` or `R3` register, which the boot ROM uses as the external memory pointers for flash/PROM or SPI memory boot, respectively.

After the processor returns from the execution of the initialization blocks, the boot ROM continues to load blocks from the location specified in the `R0` or `R3` register, which can be any `.dxe` file in the boot stream. This option requires the starting locations of specific executables within external memory. The `R0` or `R3` register must point to the 10-byte count header, as illustrated in "ADSP-BF53x and ADSP-BF561 Multi-Application (Multi-DXE) Management" on page 3-46.

ADSP-BF561 Dual-Core Application Management

A typical ADSP-BF561 dual-core application is separated into two executable files: one executable file for each core. The default linker description (`.ldf`) file for the ADSP-BF561 processor creates two separate executable files (`p0.dxe` and `p1.dxe`) and some shared memory files (`sml2.sm` and `sml3.sm`). By modifying the LDF, it is possible to create a dual-core application that combines both cores into a single `.dxe` file. This is not recommended unless the application is a simple assembly language program which does not link any C run-time libraries. When using shared memory and/or C run-time routines on both cores, it is best to generate a separate `.dxe` file for each core. The loader utility combines the contents of the shared memory files (`sml2.sm`, `sml3.sm`) only into the boot stream generated from the `.dxe` file for core A (`p0.dxe`).

By default, The boot ROM loads only one single executable before the ROM jumps to the start of core A instruction SRAM (`0xFFA0 0000`). When two `.dxe` files are loaded, a second-stage loader is used. (Or, when the -noSecondStageKernel switch is called, the loader utility combines the two `.dxe` files into one.) If the he second-stage boot loader is used, it must start at `0xFFA0 0000`. The boot ROM loads and executes the second-stage loader. A default second-stage loader is provided for each boot mode and can be customized by the user.

Unlike the initialization blocks, the second-stage loader takes full control over the boot process and never returns to the boot ROM.

The second-stage loader can use the `.dxe` byte count blocks to find specific `.dxe` files in external memory if a loader file includes the codes and data from a number of `.dxe` files.

The default second-stage loader uses the last 1024 bytes of L2 memory. The area must be reserved during booting but can be reallocated at runtime.

ADSP-BF561 Processor Memory Ranges

The on-chip boot ROM of the ADSP-BF561 processor can load a full application to the various memories of both cores. Booting is allowed to the following memory ranges. The boot ROM clears these memory ranges before booting in a new application.

- Core A
  - L1 instruction SRAM (0xFFA0 0000 – 0xFFA0 3FFF)
  - L1 instruction cache/SRAM (0xFFA1 0000 – 0xFFA1 3FFF)
  - L1 data bank A SRAM (0xFF80 0000 – 0xFF80 3FFF)
  - L1 data bank A cache/SRAM (0xFF80 4000 – 0xFF80 7FFF)
  - L1 data bank B SRAM (0xFF90 0000 – 0xFF90 3FFF)
  - L1 data bank B cache/SRAM (0xFF90 4000 – 0xFF90 7FFF)
- Core B
  - L1 instruction SRAM (0xFF60 0000 – 0xFF6 03FFF)
  - L1 instruction cache/SRAM (0xFF61 0000 – 0xFF61 3FFF)
  - L1 data bank A SRAM (0xFF40 0000 – 0xFF40 3FFF)
  - L1 data bank A cache/SRAM (0xFF40 4000 – 0xFF40 7FFF)
  - L1 data bank B SRAM (0xFF50 0000 – 0xFF50 3FFF)
  - L1 data bank B cache/SRAM (0xFF50 4000 – 0xFF50 7FFF)
- 128K of shared L2 memory (FEB0 0000 – FEB1 FFFF)
- Four banks of configurable synchronous DRAM (0x0000 0000 – (up to) 0x1FFF FFFF)

The boot ROM does not support booting to core A scratch memory (0xFFB0 0000 – 0xFFB0 0FFF) and to core B scratch memory (0xFF70 0000–0xFF70 0FFF). Data that needs to be initialized prior to runtime should not be placed in scratch memory.

ADSP-BF53x and ADSP-BF561 Multi-Application (Multi-DXE) Management

This section does not apply to the ADSP-BF535 processors.

This section describes how to generate and boot more than one `.dxe` file for the ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 and ADSP-BF561 processors. For further information about the ADSP-BF561 processors, refer to "ADSP-BF561 Dual-Core Application Management" on page 3-44.

The ADSP-BF531/BF532/BF533/BF534/ BF536/BF537/BF538/BF539 and ADSP-BF561 loader file structure and the silicon revision of 0.1 and higher allow generation and booting of multiple `.dxe` files into a single processor from external memory. As illustrated in Figure 3-20, each executable file is preceded by a 4-byte count header, which is the number of bytes within the executable, including headers. This information can be used to boot a specific `.dxe` file into the processor. The 4-byte `.dxe` count block is encapsulated within a 10-byte header to be compatible with the silicon revision 0.0. For more information, see "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Block Headers and Flags" on page 3-10.

Booting multiple executables can be accomplished by one of the following methods.

- Use the second-stage loader switch, -l userkernel.dxe. The option allows you to use your own second-stage loader.

  After the second-stage loader is booted into internal memory via

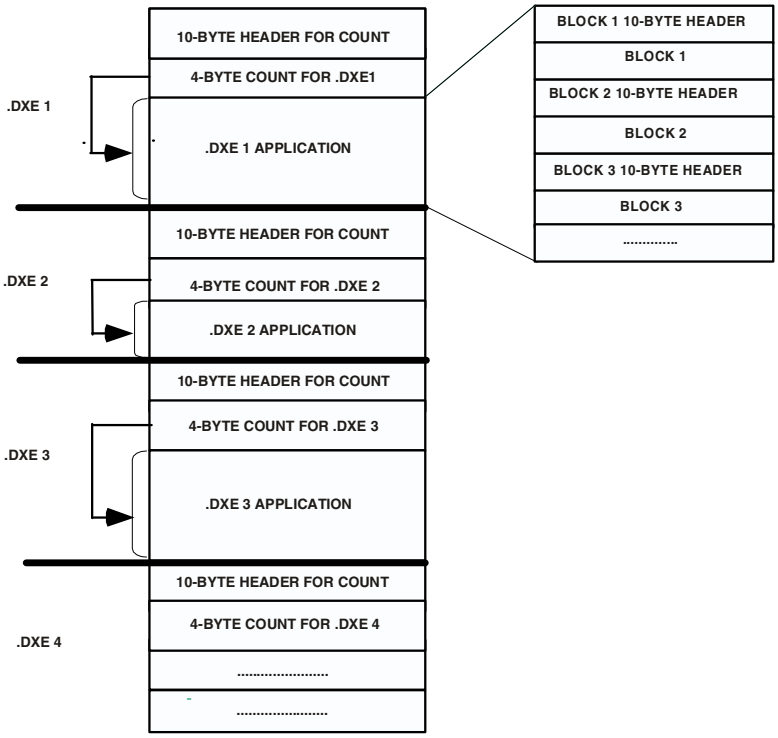

Figure 3-20. ADSP-BF531/BF32/BF33/BF534/ BF536/BF537/BF538/ BF539/BF561 Processors: Multi-Application Booting Streams

the on-chip boot ROM, the loader has full control over the boot process. Now the second-stage loader can use the `.dxe` byte counts to boot in one or more `.dxe` files from external memory.

- Use the initialization block switch, -init filename.dxe, where *`filename.dxe`* is the name of the executable file containing the initialization code. This option allows you to change the external memory pointer and boot a specific `.dxe` file via the on-chip boot ROM. On the ADSP-BF531 and ADSP-BF561 processors, the initialization code is an assembly written subroutine.

A sample initialization code is included in Listing 3-5. The R0 and R3 registers are used as external memory pointers by the on-chip boot ROM. The R0 register is for flash/PROM boot, and R3 is for SPI memory boot. Within the initialization block code, change the value of R0 or R3 to point to the external memory location at which the specific application code starts. After the processor returns from the initialization block code to the on-chip boot ROM, the on-chip boot ROM continues to boot in bytes from the location specified in the R0 or R3 register.

Listing 3-5. Initialization Block Code Example for Multiple .dxe Boot

```
#include <defBF532.h>
.SECTION program;
/*******Pre-Init Section*************************************/
     [--SP] = ASTAT;
     [--SP] = RETS;
     [--SP] = (r7:0);
     [--SP] = (p5:0);
     [--SP] = I0;[--SP] = I1;[--SP] = I2;[--SP] = I3;
     [--SP] = B0;[--SP] = B1;[--SP] = B2;[--SP] = B3;
     [--SP] = M0;[--SP] = M1;[--SP] = M2;[--SP] = M3;
     [--SP] = L0;[--SP] = L1;[--SP] = L2;[--SP] = L3;
/*************************************************************/
/*******Init Code Section*************************************
R0.H = High Address of DXE Location (R0 for flash/PROM boot,
                                     R3 for SPI boot)
R0.L = Low Address of DXE Location. (R0 for flash/PROM boot,
                                     R3 for SPI boot)
*************************************************************/
/*******Post-Init Section************************************/
     L3 = [SP++]; L2 = [SP++]; L1 = [SP++]; L0 = [SP++];
     M3 = [SP++]; M2 = [SP++]; M1 = [SP++]; M0 = [SP++];
     B3 = [SP++]; B2 = [SP++]; B1 = [SP++]; B0 = [SP++];
     I3 = [SP++]; I2 = [SP++]; I1 = [SP++]; I0 = [SP++];
```

```
      (p5:0) = [SP++];
      /* MAKE SURE NOT TO RESTORE
      R0 for flash/PROM Boot, R3 for SPI Boot */
      (r7:0) = [SP++];
      RETS = [SP++];
      ASTAT = [SP++];
/*************************************************************/
      RTS;
```

ADSP-BF531/BF532/BF533/BF534/BF536/BF537 Processor Compression Support

The loader utility for the ADSP-BF531/BF532/BF533/BF534/BF536/BF537 processors offers a loader file (boot stream) compression mechanism known as zLib. The zLib compression is supported by a third party dynamic link library, `zLib1.dll`. Additional information about the library can be obtained from the `http://www.zlib.net` Web site.

The `zLib1.dll` dynamic link library is included in VisualDSP++. The library functions perform the boot stream compression and decompression procedures when the appropriate options are selected for the loader utility. The initialization executable files with built-in decompression mechanism must perform the decompression on a compressed boot stream in a boot process. The default initialization executable files with decompression functions are included in VisualDSP++.

The loader -compression switch directs the loader utility to perform the boot stream compression from the command line. VisualDSP++ also offers a dedicated loader property page (see Figure 3-27) to manage the compression from the IDDE.

The loader utility takes two steps to compress a boot stream. First, the utility generates the boot stream in the conventional way (builds data blocks), then applies the compression to the boot stream. The decompres-

sion initialization is the reversed process: the loader utility decompresses the compressed stream first, then loads code and data into memory segments in the conventional way.

The loader utility compresses the boot stream on the `.dxe`-by-`.dxe` basis. For each input `.dxe` file, the utility compresses the code and data together, including all code and data from any associated overlay (`.ovl`) and shared memory (`.sm`) files.

Compressed Streams

Figure 3-21 illustrates the basic structure of a loader file with compressed streams.

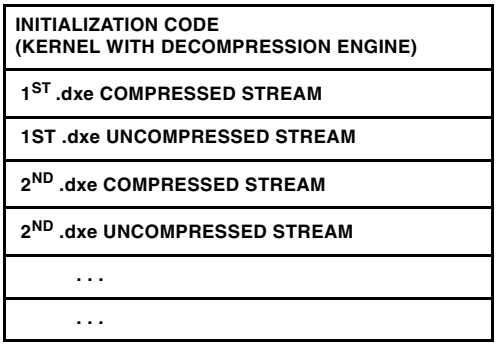

Figure 3-21. Loader File with Compressed Streams

The initialization code is on the top of the loader file. The initialization code is loaded into the processor first and is executed first when a boot process starts. Once the initialization code is executed, the rest of the stream is brought into the processor. The initialization code calls the decompression routine to perform the decompression operation on the stream, and then loads the decompressed stream into the processor's memory in the same manner a conventional boot kernel does when it encounters a compressed stream. Finally, the loader utility loads the uncompressed boot stream in the conventional way.

The Figure 3-22 illustrates the structure of a compressed block.

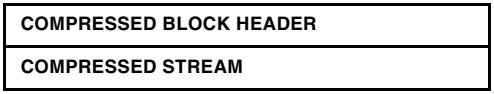

Figure 3-22. Compressed Block

Compressed Block Headers

A compressed stream always has a header, followed by the payload compressed stream. Figure 3-23 shows the structure of a compressed block header.

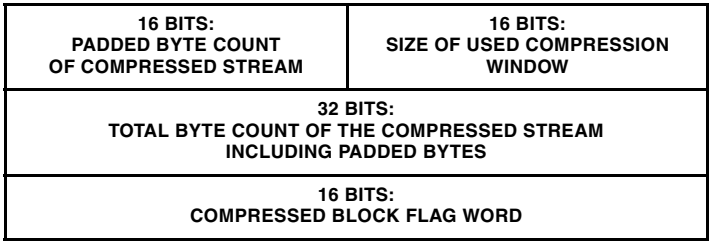

Figure 3-23. Compressed Block Header

The first 16 bits of the compressed block header hold the padded byte count of the compressed stream. The loader utility always pads the byte count if the resulting compressed stream from the loader compression engine is an odd number. The loader utility rounds up the byte count of the compressed stream to be a next higher even number. This 16-bit value is either `0x0000` or `0x0001`.

The second 16 bits of the compressed block header hold the size of the compression window, used by the loader compression engine. The value range is 8–15 bits, with the default value of 9 bits. The compression win-

dow size specifies to the compression engine a number of bytes taken from the window during the compression. The window size is the 2's exponential value.

As mentioned before, the compression/decompression mechanism for Blackfin processors utilizes the open-source lossless data-compression library zLib1. The zLib1 deflate algorithm, in turn, is a combination of a variation of Huffman coding and LZ77 compression algorithms.

LZ77 compression works by finding sequences of data that are repeated within a sliding window. As expected, with a larger sliding window, the compression algorithm is able to find more repeating sequences of data, resulting in higher compression ratios. However, technical limitations of the zLib1 decompression algorithm dictate that the window size of the decompressor must be the same as the window size of the compressor. For a more detailed technical explanation of the compression/decompression implementation on a Blackfin processor, refer to the `readme.txt` file in the `<install_path>\Blackfin\ldr\zlib\src` directory of VisualDSP++.

It is not recommended to use memory ranges used by the zlib kernel. The memory ranges used by the kernel, such as heap and static data, are defined in the LDF file, for example in `<install_path>\Blackfin\ldr\zlib\src\blkfin_zlib_init.ldf` of VisualDSP++ 5.0.

In the Blackfin implementation, the decompressor is part of the decompression initialization files (see "Decompression Initialization Files" on page 3-54). These files are built with a default decompressor window size of 9 bits (512 bytes). Thus, if you choose a non-default sliding window size for the compressor by sliding the **Compression Window Size** slider bar in the **Compression** tab (under **Load** in the **Project Options** dialog box), then the decompressor must be re-built with the newly chosen window size. For details on re-building of the decompressor init project, refer to the `readme.txt` file located in the `<install_path>\Blackfin\ldr\zlib\src` VisualDSP++ directory.

While it is true that a larger compression window size results in better compression ratios, note that there are counter factors that decrease the overall effective compression ratios with increasing window sizes for Blackfin's implementation of zlib. This is because of the limited memory resources on an embedded target, such as a Blackfin processor. For more information, refer to the `readme.txt` file in the `<install_path>\Blackfin\ldr\zlib\src` directory of VisualDSP++.

The last 16 bits of the compressed header is the flag word. The only valid compression flag assignments are shown in Figure 3-24.

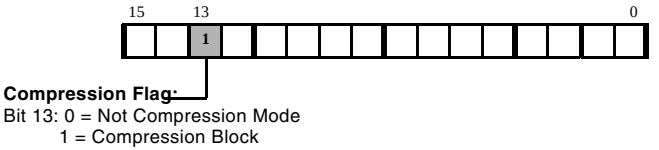

Figure 3-24. Flag Word of Compressed Block Header

Uncompressed Streams

Following the compressed streams (see Figure 3-21), the loader file includes the uncompressed streams. The uncompressed streams include application codes, conflicted with the code in the initialization blocks in the processor's memory spaces, and a final block. The uncompressed stream includes only a final block if there is no conflicted code. The final block can have a zero byte count. The final block indicates the end of the application to the initialization code.

Booting Compressed Streams

The Figure 3-25 shows the booting sequence of a loader file with compressed streams. The loader file is pre-stored in the flash memory.

1. The boot ROM is pointing to the start of the flash memory. The boot ROM reads the initialization code header and boots the initialization code.
2. The boot ROM jumps to and starts executing the initialization code.
3. (A) The initialization code scans the header for any compressed streams (see the compression flag structure in Figure 3-24). The code decompresses the streams to the decompression window (in parts) and runs the initialization kernel on the decompressed data.

   (B) The initialization kernel boots the data into various memories just as the boot ROM kernel does.
4. The initialization code sets the boot ROM to boot the uncompressed blocks and the final block (`FINAL` flag is set in the block header's flag word). The boot ROM boots the final payload, overwriting any areas used by the initialization code. Because the final flag is set in the header, the boot ROM jumps to `EVT1` (`0xFFA0 0000` for the ADSP-BF533/BF534/BF536/BF537/BF538 and ADSP-BF539 processors; `0xFFA0 8000` for the ADSP-BF531/BF532 processors) to start application code execution.

Decompression Initialization Files

As stated before, a decompression initialization `.dxe` file must be used when building a loader file with compressed streams. The decompression initialization `.dxe` file has a built-in decompression engine to decompress the compressed streams from the loader file.

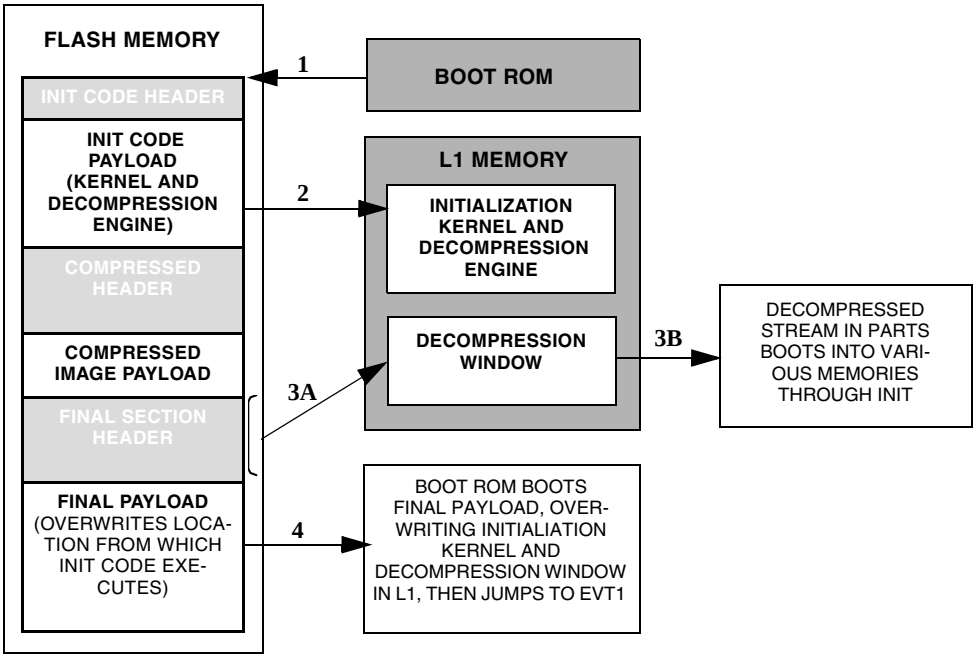
FLASH MEMORY
INIT CODE HEADER
INIT CODE PAYLOAD (KERNEL AND DECOMPRESSION ENGINE)
COMPRESSED HEADER
COMPRESSED IMAGE PAYLOAD
FINAL SECTION HEADER
FINAL PAYLOAD (OVERWRITES LOCATION FROM WHICH INIT CODE EXECUTES)
1
2
3A
4
BOOT ROM
L1 MEMORY
INITIALIZATION KERNEL AND DECOMPRESSION ENGINE
DECOMPRESSION WINDOW
3B
DECOMPRESSED STREAM IN PARTS BOOTS INTO VARIOUS MEMORIES THROUGH INIT
BOOT ROM BOOTS FINAL PAYLOAD, OVERWRITING INITIALIATION KERNEL AND DECOMPRESSION WINDOW IN L1, THEN JUMPS TO EVT1

Figure 3-25. ADSP-BF531/BF532/BF533/BF534/BF536/BF537 Compressed Stream: Booting Sequence

The decompression initialization file can be specified from the loader property page or from the loader command line via the -init filename.dxe switch. VisualDSP++ includes the default decompression initialization files, which the loader utility uses if no other initialization file is specified. The default decompression initialization file is stored in the `<install_path>\Blackfin\ldr\zlib` VisualDSP++ directory. The default decompression initialization file is built for the compression window size of 9 bits.

To use a different compression window size, build your own decompression initialization file. For details, refer to the `readme.txt` file located in the `<install_path>\Blackfin\ldr\zlib\src` directory. The size can be changed through the loader property page or the -compressWS # command-line switch. The valid range for the window size is [8, 15] bits.

ADSP-BF53x/BF561 Processor Loader Guide

Loader utility operations depend on the options, which control how the utility processes executable files. You select features such as boot modes, boot kernels, and output file formats via the options. The options are specified on the loader utility's command line or via the **Load** page of the **Project Options** dialog box in the VisualDSP++ environment. The **Load** page consists of multiple panes. When you open the **Load** page, the default loader settings for the selected processor are set already.

Option settings on the **Load** page correspond to switches displayed on the command line.

These sections describe how to produce a bootable or non-bootable loader file:

- "Using Blackfin Loader Command Line" on page 3-58
- "Using VisualDSP++ Loader" on page 3-72
- "Using VisualDSP++ Compression" on page 3-75
- "Using VisualDSP++ Second-Stage Loader for ADSP-BF535 Processors" on page 3-76
- "Using VisualDSP++ ROM Splitter" on page 3-78

Using Blackfin Loader Command Line

The ADSP-BF5xx Blackfin loader utility uses the following command-line syntax.

For a single input file:

```
elfloader inputfile -proc processor [-switch …]
```

For multiple input files:

```
elfloader inputfile1 inputfile2 … -proc processor [-switch …]
```

where:

- *inputfile*—Name of the executable (`.dxe`) file to be processed into a single boot-loadable or non-bootable file. An input file name can include the drive and directory. For multiprocessor or multi-input systems, specify multiple input `.dxe` files. Put the input file names in the order in which you want the loader utility to process the files. Enclose long file names within straight quotes, "`long file name`".
- `-proc` *processor*—Part number of the processor (for example, `-proc ADSP-BF531`) for which the loadable file is built. Provide a processor part number for every input `.dxe` if designing multiprocessor systems.
- *-switch …*—One or more optional switches to process. Switches select operations and modes for the loader utility.

Command-line switches may be placed on the command line in any order, except the order of input files for a multi-input system. For a multi-input system, the loader utility processes the input files in the order presented on the command line.

File Searches

File searches are important in loader processing. The loader utility supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur as described on page 1-17.

File Extensions

Some loader switches take a file name as an optional parameter. Table 3-9 lists the expected file types, names, and extensions.

Table 3-9. File Extensions

Extension	File Description
`.dxe`	Loader input files, boot kernel files, and initialization files
`.ldr`	Loader output file
`.knl`	Loader output files containing kernel code only when two output files are selected

In some cases the loader utility expects the overlay input files with the file extension of `.ovl`, shared memory input files with the extension of `.sm` or both, but does not expect those files to appear on a command line or on the **Load** property page. The loader utility finds these files in the directory of the associated `.dxe` files, in the current working directory, or in the directory specified in the `.ldf` file.

Blackfin Loader Command-Line Switches

A summary of the Blackfin loader command-line switches appears in Table 3-10.

Table 3-10. Blackfin Loader Command-Line Switch Summary

Switch	Description

`-b prom`
`-b flash`
`-b spi`
`-b spislave`
`-b UART`
`-b TWI`
`-b FIFO`
	The `-b {prom|flash|spi|spislave|UART|TWI|FIFO}` switch specifies the boot mode and directs the loader utility to prepare a boot-loadable file for the specified boot mode. Valid boot modes include PROM, flash, SPI, SPI slave, UART, TWI, and FIFO.
	
	SPI slave, UART, and TWI boot modes are for the ADSP-BF531/BF532/BF533/BF534/BF536/BF537, and ADSP-BF538/BF539 processors.
	
	FIFO boot mode is for the ADSP-BF534/BF536 and ADSP-BF537 processors of silicon revision 0.4 or newer only.
	If `-b` does not appear on the command line, the default is `-b flash`.

`-baudrate #`	The `-baudrate #` switch accepts a baud rate for SPI booting only. Valid baud rates and corresponding values (#) are: • `500K` – 500 kHz, the default value • `1M` – 1 MHz • `2M` – 2 MHz Boot kernel loading supports an SPI baud rate up to 2 MHz. Applies to the ADSP-BF535 processors only.

Table 3-10. Blackfin Loader Command-Line Switch Summary (Cont'd)

Switch	Description
`-compression`	The `-compression` switch directs the loader utility to compress the boot stream; see "ADSP-BF531/BF532/BF533/BF534/BF536/BF537 Processor Compression Support" on page 3-49. Either a default or user initialization `.dxe` file with decompression code must be provided for `-compression`. This switch is for flash/PROM boot modes only and does not apply to the ADSP-BF535, ADSP-BF538, ADSP-BF539, or ADSP-BF561 processors.
`-compressWS #`	The `-compressWS #` switch specifies a compression window size in bytes. The number is a 2's exponential value to be used by the compression engine. The valid values are `[8,15]` bits, with the default of `9` bits. This switch is for flash/PROM boot modes only and does not apply to the ADSP-BF535, ADSP-BF538, ADSP-BF539, or ADSP-BF561 processors.
`-dmawidth #`	The `-dmawidth {8|16}` switch specifies a DMA width (in bits) to the loader utility. For FIFO boot mode, `16` is the only DMA width. For other boot modes, all DMA widths are valid with the default of eight. Does not apply to the ADSP-BF535 or ADSP-BF561 processors.
`-enc` *`dll_filename`*	The `-enc` *`dll_filename`* switch encrypts the data stream from the application input `.dxe` files with the encryption algorithms in the dynamic library file *`dll_filename`*. If the *`dll_filename`* parameter does not appear on the command line, the encryption algorithm from the default ADI's file is used.

Table 3-10. Blackfin Loader Command-Line Switch Summary (Cont'd)

Switch	Description

`-f hex`
`-f ASCII`
`-f binary`
`-f include`
	The `-f {hex|ASCII|binary|include}` switch specifies the format of a boot-loadable file (Intel hex-32, ASCII, binary, include). If the `-f` switch does not appear on the command line, the default boot mode format is hex for flash/PROM and ASCII for SPI, SPI slave, UART, and TWI.

`-ghc #`	The `-ghc #` switch specifies a 4-bit value (global header cookie) for bits 31–28 of the global header (see Table 3-7 on page 3-38). Applies to the ADSP-BF561 processors only.
`-h` or `-help`	The `-h[elp]` switch invokes the command-line help, outputs a list of command-line switches to standard output, and exits. By default, the `-h` switch alone provides help for the loader driver. To obtain a help screen for your target Blackfin processor, add the `-proc` switch to the command line. For example: type `elfloader -proc ADSP-BF535 -h` to obtain help for the ADSP-BF535 processor.
`-HoldTime #`	The `-HoldTime #` switch allows the loader utility to specify a number of hold time cycles for flash/PROM boot. The valid values (#) are from `0` through `3`. The default value is `3`. Applies to the ADSP-BF535 processors only.
`-init` *`filename.dxe`*	The `-init` *`filename.dxe`* switch directs the loader utility to include the initialization code from the named file. The loader utility places the code from the initialization sections of the specified `.dxe` file in the boot stream. The kernel loads the code and then calls it. It is the responsibility of the code to save/restore state/registers and then perform an RTS back to the kernel. Does not apply to the ADSP-BF535 processors.

Table 3-10. Blackfin Loader Command-Line Switch Summary (Cont'd)

Switch	Description

`-kb prom`
`-kb flash`
`-kb spi`
`-kb spislave`
`-kb UART`
`-kb TWI`
`-kb FIFO`
	The `-kb {prom|flash|spi|spislave|UART|TWI|FIFO}` switch specifies the boot mode (PROM, flash, SPI, SPI slave, UART, TWI, or FIFO) for the boot kernel output file if you generate two output files from the loader utility: one for the boot kernel and another for user application code.
	The `spislave`, `UART`, and `TWI` parameters are applicable to the ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538 and ADSP-BF539 processors only. The `FIFO` parameter applies to the ADSP-BF534/BF536/BF537 processors of silicon revision 0.4 or newer.
	 The `-kb` switch must be used in conjunction with the `-o2` switch.
	If the `-kb` switch is absent from the command line, the loader utility generates the file for the boot kernel in the same boot mode as used to output the user application program.

`-kf hex`
`-kf ascii`
`-kf binary`
`-kf include`
	The `-kf {hex|asci|binary|include}` switch specifies the output file format (hex, ASCII, binary, or include) for the boot kernel if you output two files from the loader utility: one for the boot kernel and one for user application code.
	 The `-kf` switch must be used in conjunction with the `-o2` switch.
	If the `-kf` switch is absent from the command line, the loader utility generates the file for the boot kernel in the same format as for the user application program.

`-kenc` *`dll_filename`*	The `-kenc` *`dll_filename`* switch specifies the user encryption dynamic library file for the encryption of the data stream from the kernel file. If the filename parameter does not appear on the command line, the encryption algorithm from the default ADI's file is used.
`-kp #`	The `-kp #` switch specifies a hex flash/PROM output start address for the kernel code. A valid value is between `0x0` and `0xFFFFFFFF`. The specified value is ignored when no kernel or/and initialization code is included in the loader file.

Table 3-10. Blackfin Loader Command-Line Switch Summary (Cont'd)

Switch	Description
`-kWidth #`	The `-kWidth #` switch specifies the width of the boot kernel output file when there are two output files: one for the boot kernel and one for user application code. Valid values are: • Eight or `16` for PROM or flash boot kernel • `16` for FIFO boot kernel • Eight for SPI and other boot kernels If this switch is absent from the command line, the default file width is: • the `-width` parameter for flash/PROM boot mode • `16` for FIFO boot mode • Eight when booting from SPI and other boot modes The `-kWidth #` switch must be used in conjunction with the `-o2` switch.
`-l userkernel.dxe`	The `-l userkernel.dxe` switch specifies the user boot kernel file. The loader utilizes the user-specified kernel and ignores the default boot kernel if there is one. Currently, only the ADSP-BF535 processors have default kernels.
`-M`	The `-M` switch generates make dependencies only, no output file is generated.
`-maskaddr #`	The `-maskaddr #` switch masks all EPROM address bits above or equal to #. For example, `-maskaddr 29` (default) masks all the bits above and including `A29` (`AND`ed by `0x1FFF FFFF`). For example, `0x2000 0000` becomes `0x0000 0000`. The valid #s are integers `0` through `32`, but based on your specific input file, the value can be within a subset of [0, 32]. The `-maskaddr #` switch requires `-romsplitter` and affects the ROM section address only.
`-MaxBlockSize #`	The `-MaxBlockSize #` switch specifies the maximum block byte count, which must be a multiple of 16.

Table 3-10. Blackfin Loader Command-Line Switch Summary (Cont'd)

Switch	Description
`-MaxZeroFillBlockSize #`	The `-MaxZeroFillBlockSize #` switch specifies the maximum block byte count for zero-filled blocks. The valid values are from `0x0` to `0xFFFFFFF0`, and the default value matches `-MaxBlockSize #`.
`-MM`	The `-MM` switch generates make dependencies while producing the output files.
`-Mo` *`filename`*	The `-Mo` *`filename`* switch writes make dependencies to the named file. Use the `-Mo` switch with either `-M` or `-MM`. If `-Mo` is not present, the default is a `<stdout>` display.
`-Mt` *`filename`*	The `-Mt` *`filename`* switch specifies the make dependencies target output file. Use the `-Mt` switch with either `-M` or `-MM`. If `-Mt` is not present, the default is the name of the input file with an `.ldr` extension.
`-no2kernel`	The `-no2kernel` switch produces the output file without the boot kernel but uses the boot-strap code from the internal boot ROM. The boot stream generated by the loader utility is different from the one generated by the boot kernel. Applies to the ADSP-BF535 processors only.
`-noFinalBlock`	The `-noFinalBlock` switch directs the loader utility not to make a special final block for TWI boot. Applies to the ADSP-BF537 processors only.
`-noFinalTag`	The `-noFinalTag` switch directs the loader utility not to set the final block tag for the first `.dxe` file. As a result, the boot process continues with code from the second `.dxe` file, following the first file. Applies to the ADSP-BF56x processors only.

Table 3-10. Blackfin Loader Command-Line Switch Summary (Cont'd)

Switch	Description
`-noInitCode`	The `-noInitCode` switch directs the loader utility not to expect an initialization input file even though an external memory section is present in the input `.dxe` file. Applies to the ADSP-BF531/BF532/BF533, ADSP-BF534/BF536/BF537/BF538/BF539 processors only.
`-noSecondStageKernel`	The `-noSecondStageKernel` switch directs the loader utility not to include a default second-stage kernel into the loader stream. Applies to the ADSP-BF56x processors only.
`-o` *`filename`*	The `-o` *`filename`* switch directs the loader utility to use the specified file as the name of the loader utility's output file. If the *`filename`* is absent, the default name is the root name of the input file with an `.ldr` extension.
`-o2`	The `-o2` switch produces two output files: one for the init block (if present) and boot kernel and one for user application code. To have a different format, boot mode, or output width from the application code output file, use the `-kb -kf -kwidth` switches to specify the boot mode, the boot format, and the boot width for the output kernel file, respectively. Do not combine the `-o2` switch with `-nokernel` on the ADSP-BF535 processors. Combine `-o2` with `-l` *`filename`* and/or `-init` *`filename`* on the ADSP-BF531/BF532/BF533, ADSP-BF534/BF536/BF537/BF538/BF539, ADSP-BF561 processors.
`-p #`	The `-p #` switch specifies a hex flash/PROM output start address for the application code. A valid value is between `0x0` and `0xFFFFFFFF`. A specified value must be greater than that specified by `-kp` if both kernel and/or initialization and application code are in the same output file (a single output file).

Table 3-10. Blackfin Loader Command-Line Switch Summary (Cont'd)

Switch	Description

`-pFlag #`
`-pFlag PF#`
`-pFlag PG#`
`-pFlag PH#`
	The `-pflag {#|PF#|PG#|PH#}` switch specifies a 4-bit hex value for a strobe (programmable flag) or for one of the ports: `F`, `G`, or `H`. There is no default value. The value is dynamic and varies with processor, silicon revision, boot mode, and width. The loader generates warnings for illegal combinations.
	Table 3-11, Table 3-12, and Table 3-11 show the valid values for the `-pFlag` switch.
	 Applies to the ADSP-BF531/BF532/BF533/ BF534, ADSP-BF536/BF537/BF538/BF539, and ADSP-BF561 processors only.

`-proc` *processor*	The `-proc` *processor* switch specifies the target processor. The *processor* can be one of the following: `ADSP-BF531`, `ADSP-BF532`, `ADSP-BF533`, `ADSP-BF534`, `ADSP-BF535`, `ADSP-BF536`, `ADSP-BF537`, `ADSP-BF538`, `ADSP-BF539`, `ADSP-BF561`.
`-romsplitter`	The `-romsplitter` switch creates a non-bootable image only. This switch overwrites the `-b` switch and any other switch bounded by the boot mode. In the `.ldf` file, declare memory segments to be 'split' as type `ROM`. The splitter skips `RAM` segments, resulting in an empty file if all segments are declared as `RAM`. The `-romsplitter` switch supports hex and ASCII formats.
`-ShowEncryptionMessage`	The `-ShowEncryptionMessage` switch displays a message returned from the encryption function.

Table 3-10. Blackfin Loader Command-Line Switch Summary (Cont'd)

Switch	Description
`-si-revision #|none|any`	The `-si-revision {#|none|any}` switch provides a silicon revision of the specified processor. The switch parameter represents a silicon revision of the processor specified by the `-proc` *`processor`* switch. The parameter takes one of three forms: • The `none` value indicates that the VisualDSP++ ignores silicon errata. • The # value indicates one or more decimal digits, followed by a point, followed by one or two decimal digits. Examples of revisions are: `0.0`; `1.12`; `23.1`. Revision `0.1` is distinct from and "lower" than revision `0.10`. The digits to the left of the point specify the chip tapeout number; the digits to the right of the point identify the metal mask revision number. The number to the right of the point cannot exceed decimal `255`. • The `any` value indicates that VisualDSP++ produces an output file that can be run at any silicon revision. The switch generates either a warning about any potential anomalous conditions or an error if any anomalous conditions occur. In the absence of the silicon revision switch, the loader utility selects the greatest silicon revision it is aware of, if any.
`-v`	The `-v` switch directs the loader utility to output verbose loader messages and status information as the loader processes files.
`-waits #`	The `-waits #` switch specifies the number of wait states for external access. Valid inputs are `0` through `15`. Default is `15`. Wait states apply to the flash/PROM boot mode only. Applies to the ADSP-BF535 processors only.

Table 3-10. Blackfin Loader Command-Line Switch Summary (Cont'd)

Switch	Description
`-width #`	The `-width #` switch specifies the loader output file's width in bits. Valid values are `eight` and `16`, depending on the boot mode. The default value is `16` for FIFO boot mode and `eight` for all other boot modes. On the ADSP-BF535 processors, the switch has no effect on boot kernel code processing. The loader utility processes the kernel in eight-bit widths regardless of the output width selection. • For flash/PROM booting, the size of the output file depends on the `-width #` switch. • For FIFO booting, the only available width is `16`. • For SPI booting, the size of the output `.ldr` file is the same for both `-width 8` and `-width 16`. The only difference is the header information.
`-ZeroPadForced`	The `-ZeroPadForced` switch forces the loader utility to pad each data byte with a zero byte for 16-bit output. Use this switch only if your system requires zero padding in a loader file. Use this switch with caution: arbitrating pad data with zeros can cause the loader file to fail. The loader utility performs default zero padding automatically in general. Applies to the ADSP-BF531/BF532/BF533/BF534, ADSP-BF536/BF537/BF538/BF539 processors only.

Table 3-11. -pFlag Values for ADSP-BF531/BF532/BF533 Processors[1]

Silicon Revision	0.0–0.2		0.3–0.5	
Width	8	16	8	16
Flash boot mode	NONE	NONE	NONE	NONE
SPI boot mode	NONE		NONE	
SPI slave boot mode			1-15 PF1-15	

1 The ADSP-BF531/BF532/BF533 processors always have the `RESVECT` bit (bit 2 in the block header flag word) cleared.

Table 3-12. -pFlag Values for ADSP-BF534/BF536/BF537[1]

Silicon Revision	0.0		0.1–0.2		0.3	
Width	8	16	8	16	8	16
Flash boot mode	NONE	NONE	NONE PF0-15 PG0-15 PH0-15	NONE PF0-15 PG0-15 PH0-15	NONE PF0-15 PG0-15 PH0-15	NONE PF0-15 PG0-15 PH0-15
SPI boot mode	NONE		NONE PF0-9 PF15 PG0-15 PH0-15		NONE PF0-9 PF15 PG0-15 PH0-15	
SPI slave boot mode	1-15 PF1-15		NONE PF0-10 PF15 PG0-15 PH0-15		NONE PF0-10 PF15 PG0-15 PH0-15	
TWI boot mode	NONE		NONE PF0-15 PG0-15 PH0-15		NONE PF0-15 PG0-15 PH0-15	
TWI slave boot mode	NONE		NONE PF0-15 PG0-15 PH0-15		NONE PF0-15 PG0-15 PH0-15	
UART boot mode	2-15 PF2-15		NONE PF2-15 PG0-15 PH0-15		NONE PF2-15 PG0-15 PH0-15	
FIFO boot mode						NONE PF0 PF2-15 PG0-15 PH0-15

1 The ADSP-BF534/BF536/BF537 processors always have the RESVECT bit (bit 2 in the block header flag word) set.

Table 3-13. -pFlag Values for ADSP-BF538/BF539 Processors[1]

Silicon Revision	0.0–0.3	
Width	8	16
Flash boot mode	NONE	NONE
SPI boot mode	NONE	
SPI slave boot mode	1-15 PF1-15	

1 The ADSP-BF538/BF539 processors always have the RESVECT bit (bit 2 in the block header flag word) set.

Using VisualDSP++ Loader

After selecting a **Loader file** (**.ldr**) as the project output type for your application on the **Application Settings** page in the VisualDSP++ **Project Wizard**, modify the default load settings.

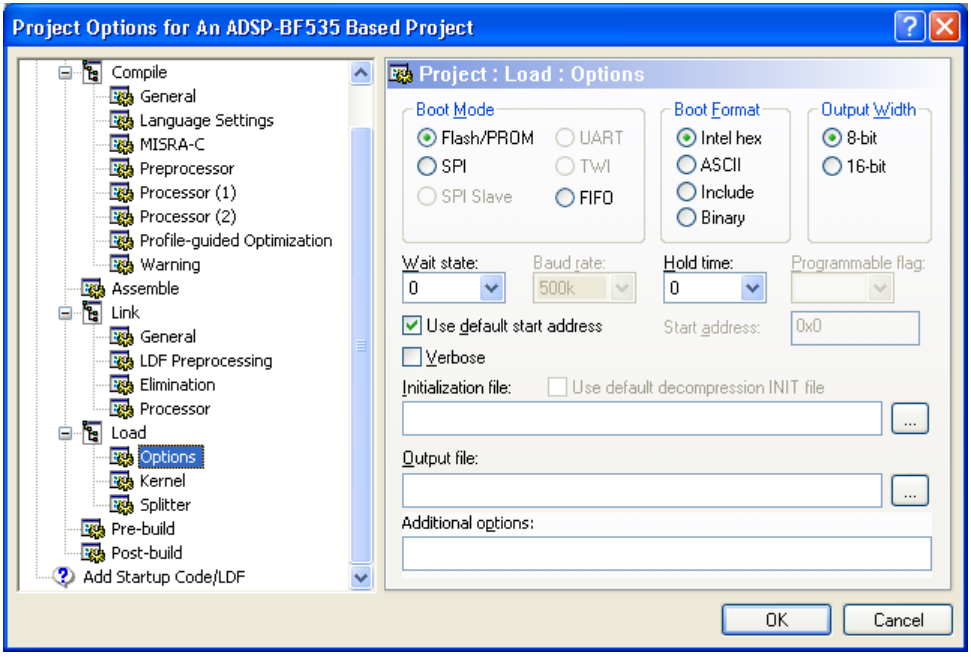

Figure 3-26. Project: Load: Options Page for ADSP-BF535 Processors

The **Load** control in the **Project** tree control consists of multiple pages. When you open the **Load: Options** page (also called loader property page), view the default load settings for the selected processor. As an example, Figure 3-26 shows the ADSP-BF535 processor's default load settings for PROM booting. The dialog box options are equivalent to the command-line switches. Refer to "Blackfin Loader Command-Line Switches" on page 3-60 for more information about the switches.

Using the page controls, select or modify the load settings. Table 3-14 describes each load control and corresponding setting. When satisfied with the settings, click **OK** to complete the load setup.

Table 3-14. Base Load Page Settings for ADSP-BF53x/BF561 Processors

Setting	Description
Load	Selections for the loader utility. The options are: • **Options** – default boot options (this section) • **Compression** – specification for zLib compression; applies to the ADSP-BF531/BF532/BF533/BF534, ADSP-BF536, and ADSP-BF537 processors (see page 3-49). For the ADSP-BF535 processor based projects, the compression is not available. • **Kernel** – specification for a second-stage loader. Can be used to override the default boot kernel if there is one by default, as on the ADSP-BF535 processors (see page 3-76). • **Splitter** – specification for the no-boot mode (see page 3-78) If you do not use the boot kernel for the ADSP-BF535 processors, the **Kernel** page appears with all kernel option fields grayed out. The loader utility does not search for the boot kernel if you boot from the on-chip ROM by setting the `-no2kernel` command-line switch as described on page 3-65. For the ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539/BF561 processors, which do not have software boot kernels by default, select the boot kernel to use one.
Boot mode	Specifies flash/PROM, SPI, SPI slave, UART, TWI, or FIFO as a boot source
Boot format	Specifies Intel hex, ASCII, include, or binary format
Output width	Specifies eight or 16 bits. If `BMODE = 01` or `001` and flash/PROM is 16-bit wide, the 16-bit option must be selected.
Wait state	Specifies the number of wait states for external access (`0`–`15`). The selection is active for the ADSP-BF535 processors. For the ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539/BF561 processors, the field is grayed out.

Table 3-14. Base Load Page Settings for ADSP-BF53x/BF561 Processors

Setting	Description
Baud rate	Specifies a baud rate for SPI booting (500 kHz, 1 MHz, and 2 MHz). The selection is active for the ADSP-BF535 processors. For the ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539/ BF561 processors, the field is grayed out.
Hold time	Specifies the number of the hold time cycles for flash/PROM boot (0–3). The selection is active for the ADSP-BF535 processors. For the ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539/ BF561 processors, the field is grayed out.
Programmable flag	Same as the `-pFlag` command-line switch—selects a programmable flag number (0–15) for a strobe or for a port. The box is active for the ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 processors. Valid values are listed in Table 3-11 through Table 3-13. The `NONE` option also is available; when chosen, no `-pFlag` switch appears on the command line. Verify the programmable flag setting whenever the processor, silicon revision, boot mode, or width is changed.
Use default start address	Specifies the default flash/PROM output start address in hex format for the application code.
Start address	Specifies a flash/PROM output start address in hex format for the application code.
Verbose	Generates status information as the loader utility processes the files.
Initialization file	Directs the loader utility to include the initialization file (init code).
Use default decompression INIT file	Directs the loader utility to include the default decompression initialization file (init code). The initialization file selection is active for the ADSP-BF531/BF532/BF533, and ADSP-BF561 processors. For the ADSP-BF535 processors, the field is grayed out.
Output file	Names the loader utility's output file.
Additional options	Specifies additional loader switches. You can specify additional input files for a multi-input system. Type the input file names with the paths if the files are not in the current working directory, separate two file names with a space in order for the loader utility to retrieve the files. **Note:** The loader utility processes the input files in the order in which the files appear on the command line, generated from the property page.

Using VisualDSP++ Compression

If you develop an ADSP-BF531/BF532/BF533/BF534/BF536 or ADSP-BF537 processor based application, you can select **Compression** under **Load** in the **Project Options** tree control to set parameters for zLib compression.

To enable compression, select **Enable Compression**. You can select the **Compression window size** (**~2**N**), **Retain kernel after boot**, and **Compress Overlays** options. The dialog box options are equivalent to command-line switches. See "ADSP-BF531/BF532/BF533/BF534/BF536/BF537 Processor Compression Support" on page 3-49 for more information.

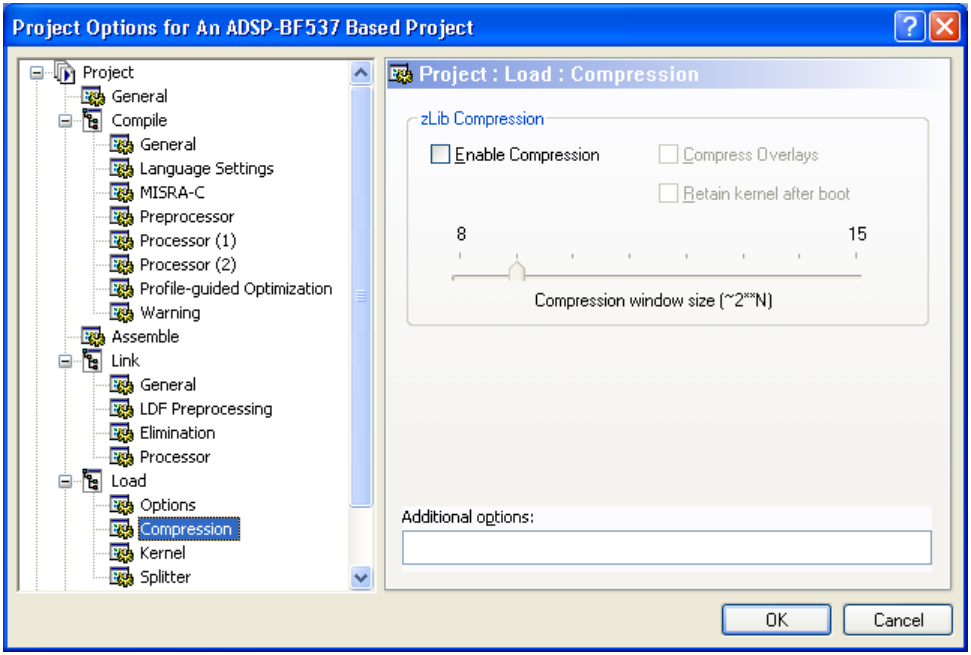

Figure 3-27. Project: Load: Compression Page for ADSP-BF537 Processors

Using VisualDSP++ Second-Stage Loader for ADSP-BF535 Processors

If you use a second-stage loader, select **Kernel** under **Load** in the **Project Options** tree control. The page shows the default settings for a loader file that includes a second-stage loader.

Unless you develop an application for the ADSP-BF535 processor, most of the options on the **Kernel** page are grayed out. Figure 3-28 shows a sample **Kernel** page with options set for an ADSP-BF535 Blackfin processor.

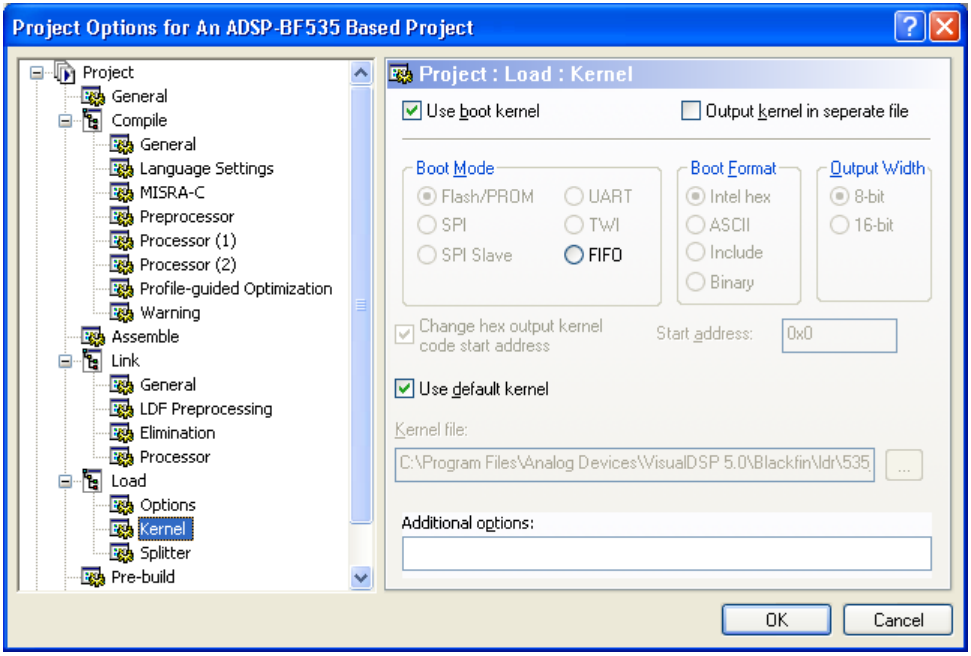

Figure 3-28. Project: Load: Kernel Page for ADSP-BF535 Processors

To create a loader file which includes a second-stage loader:

1. Select **Options** (under **Load**) to set up base load options (see "Using VisualDSP++ Loader" on page 3-72).
2. Select **Kernel** (under **Load**) to set up the second-stage loader options (Figure 3-28).
3. On the **Kernel** page, select **Use boot kernel**.
4. In **Kernel file**, enter the name of the second-stage loader file (`.dxe`).

   The **Use default kernel** option is available for the ADSP-BF535 and grayed out for the ADSP-BF531/BF532/BF533/ BF534/BF536/BF537/BF538/BF539/BF561 processors. In case of an ADSP-BF535 processor, choose between the default or user second-stage loader file. The following default second-stage loaders are currently available for the ADSP-BF535 processors.

Boot Mode	Second -Stage Loader File
Eight-bit flash/PROM	`535_prom8.dxe`
16-bit flash/PROM	`535_prom16.dxe`
SPI	`535_spi.dxe`

For the ADSP-BF531/BF532/BF533/BF534/ BF536/BF537/BF538/BF539/BF561 processors, no second-stage loaders are required; hence, no default kernel files are provided. You can supply your own second-stage loader file if so desired (steps 3 and 4).

5. To produce two output files, select the **Output kernel in separate file** check box. This option allows to boot the second-stage loader with an initialization code (if any) from one source and the appli-

cation code from another source. You can specify the kernel output file options, such as the **Boot Mode** (source), **Boot Format**, and **Output Width**.

6. Select **Change hex output kernel code start address** to specify the **Start address** in hex format for the second-stage loader code. This option allows you to place the second-stage loader file at a specific location within the flash/PROM.

7. Click **OK** to complete the loader setup.

Using VisualDSP++ ROM Splitter

Unlike the loader utility, the splitter does not format the application data when transforming a `.dxe` file to an `.ldr` file. It emits raw data only. Whether data and/or instruction segments are processed by the loader or by the splitter utility depends upon the LDF's `TYPE()` command. Sections declared with `TYPE(RAM)` are consumed by the loader utility, and sections declared by `TYPE(ROM)` are consumed by the splitter.

Figure 3-29 shows a sample **Load: Splitter** page with ROM splitter options. With the **Enable ROM splitter** box unchecked, only `TYPE(RAM)` sections are processed and all `TYPE(ROM)`) segments are ignored by the loader utility. If the box is checked, `TYPE(RAM)` sections are ignored, and `TYPE(ROM)` sections are processed by the splitter utility.

The **Mask Address** field masks all EPROM address bits above or equal to the number specified. For example, **Mask Address** = `29` (default) masks all bits above and including `A29` (`AND`ed by `0x1FFF FFFF`). Thus, `0x2000 0000` becomes `0x0000 0000`. The valid numbers are integers `0` through `32` but, based on your specific input file, the value can be within a subset of [`0`, `32`].

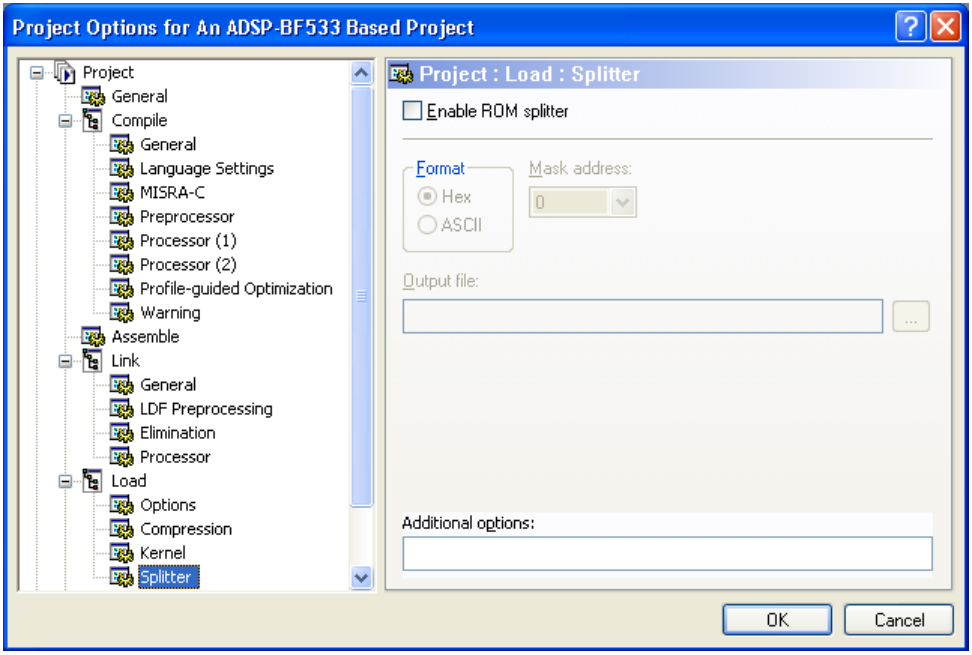

Figure 3-29. Project: Splitter Page for ADSP-BF533 Processors

ADSP-BF535 and ADSP-BF531/BF532/BF533/BF534/ BF536/BF537/BF538/BF539 Processor No-Boot Mode

The hardware settings of `BMODE = 000` for the ADSP-BF535 processors or `BMODE = 00` for ADSP-BF531, ADSP-BF532, and ADSP-BF533 processors select the no-boot option. In this mode of operation, the on-chip boot kernel is bypassed after reset, and the processor starts fetching and

executing instructions from address 0x2000 0000 in the asynchronous memory bank 0. The processor assumes 16-bit memory with valid instructions at that location.

To create a proper .ldr file that can be burned into either a parallel flash or EPROM device, you must modify the standard LDF file in order for the reset vector to be located accordingly. The following code fragments (Listing 3-6 and Listing 3-7) illustrate the required modifications in case of an ADSP-BF533 processor.

Listing 3-6. Section Assignment (LDF File) Example

```
MEMORY
{
  /* Off-chip Instruction ROM in Async Bank 0 */
  MEM_PROGRAM_ROM { TYPE(ROM) START(0x20000000) END(0x2009FFFF)
WIDTH(8) }
  /* Off-chip constant data in Async Bank 0  */
  MEM_DATA_ROM    { TYPE(ROM) START(0x200A0000) END(0x200FFFFF)
WIDTH(8) }
  /* On-chip SRAM data, is not booted automatically */
  MEM_DATA_RAM    { TYPE(RAM) START(0xFF903000) END(0xFF907FFF)
WIDTH(8) }
```

Listing 3-7. ROM Segment Definitions (LDF File) Example

```
PROCESSOR p0
{
  OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

  SECTIONS
  {
    program_rom
    {
```

```
        INPUT_SECTION_ALIGN(4)
        INPUT_SECTIONS( $OBJECTS(rom_code) )
    } >MEM_PROGRAM_ROM
    data_rom
    {
        INPUT_SECTION_ALIGN(4)
        INPUT_SECTIONS( $OBJECTS(rom_data) )
     } >MEM_DATA_ROM
    data_sram
    {
         INPUT_SECTION_ALIGN(4)
         INPUT_SECTIONS( $OBJECTS(ram_data) )
    } >MEM_DATA_RAM
```

With the LDF file modified this way, the source files can now take advantage of the newly-introduced sections, as in Listing 3-8.

Listing 3-8. Section Handling (Source File) Example

```
.SECTION rom_code;
_reset_vector: l0 = 0;
               1 = 0;
               l2 = 0;
               l3 = 0;
               /* continue with setup and application code */
               /* . . . */
.SECTION rom_data;
.VAR myconst x = 0xdeadbeef;
               /* . . . */
.SECTION ram_data;
.VAR myvar y; /* note that y cannot be initialized automatically */
```

4 LOADER FOR ADSP-2106X/21160 SHARC PROCESSORS

This chapter explains how the loader utility (`elfloader.exe`) is used to convert executable (`.dxe`) files into boot-loadable files for the ADSP-21060, ADSP-21061, ADSP-21062, ADSP-21065L, and ADSP-21160 SHARC processors.

Refer to "Introduction" on page 1-1 for the loader utility overview; the introductory material applies to all processor families. Refer to "Loader for ADSP-21161 SHARC Processors" on page 5-1 for information about the ADSP-21161 processors. Refer to "Loader for ADSP-2126x/2136x/2137x/2146x SHARC Processors" on page 6-1 for information about the ADSP-2126x and ADSP-2136x processors.

Loader operations specific to the ADSP-2106x/21160 SHARC processors are detailed in the following sections.

- "ADSP-2106x/21160 Processor Booting" on page 4-2
  Provides general information about various booting modes, including information about boot kernels.
- "ADSP-2106x/21160 Processor Loader Guide" on page 4-25
  Provides reference information about the loader utility's graphical user interface, command-line syntax, and switches.

ADSP-2106x/21160 Processor Booting

The ADSP-2106x/21160 processors support three boot modes: EPROM, host, link port, and no-boot (see Table 4-3 and Table 4-4 on page 4-5). Boot-loadable files for these modes pack boot data into 48-bit instructions and use an appropriate DMA channel of the processor's DMA controller to boot-load the instructions.

The ADSP-2106x processors use DMA channel 6 (`DMAC6`) for booting. The ADSP-21160 processors use `DMAC8` for link port booting and `DMAC10` for the host and EPROM booting.

- When booting from an EPROM through the external port, the ADSP-2106x/21160 processor reads boot data from an 8-bit external EPROM.
- When booting from a host processor through the external port, the ADSP-2106x/21160 processor accepts boot data from a 8- or 16-bit host microprocessor.
- When booting through the link port, the ADSP-2106x/21160 processor receives boot data as 4-bit wide data in link buffer 4.
- In no-boot mode, the ADSP-2106x/21160 processor begins executing instructions from external memory.

Software developers who use the loader utility should be familiar with the following operations.

- "Power-Up Booting Process" on page 4-3
- "Boot Mode Selection" on page 4-5
- "ADSP-2106x/21160 Boot Modes" on page 4-7
- "ADSP-2106x/21160 Boot Kernels" on page 4-16
- "ADSP-2106x/21160 Interrupt Vector Table" on page 4-22

- "ADSP-2106x/21160 Multi-Application (Multi-DXE) Management" on page 4-23
- "ADSP-2106x/21160 Processor ID Numbers" on page 4-24

Power-Up Booting Process

The ADSP-2106x and ADSP-21160 processors include a hardware feature that boot-loads a small, 256-instruction program into the processor's internal memory after power-up or after the chip reset. These instructions come from a program called boot kernel. When executed, the boot kernel facilitates booting of user application code. The combination of the boot kernel and application code comprise the boot-loadable (`.ldr`) file.

At power-up, after the chip reset, the booting process includes the following steps.

1. Based on the boot type, an appropriate DMA channel is automatically configured for a 256-instruction (48-bit) transfer. This transfer boot-loads the boot kernel program into the processor memory. DMA channels used by the various processor models are shown in Table 4-1.

Table 4-1. ADSP-2106x/21160 Processor DMA Channels

Processor	PROM Booting	Host Booting	Link Booting
ADSP-21060	`DMAC6` (See Table 4-8)	`DMAC6` (See Table 4-8)	`DMAC6`
ADSP-21061			Not supported
ADSP-21062			`DMAC6`
ADSP-21065L	`DMAC8` (`DMAC0` programs `DMAC8`; see Table 4-8)	`DMAC8` (`DMAC0` programs `DMAC8`; see Table 4-8)	Not supported
ADSP-21160	`DMAC10` (See Table 4-9)	`DMAC10` (See Table 4-9)	`DMAC8`

2. The boot kernel runs and loads the application executable code and data.
3. The boot kernel overwrites itself with the first 256 words of the application at the end of the booting process. After that, the application executable code begins to execute from locations `0x20000` (ADSP-21060/61/62), `0x8000` (ADSP-21065L), and `0x40000` (ADSP-21160). The start addresses and reset vector addresses are summarized in Table 4-2.

Table 4-2. ADSP-2106x/21160 Processor Start Addresses

Processor	Start Address	Reset Vector Address[1]
ADSP-21060	`0x20000`	`0x20004`
ADSP-21061	`0x20000`	`0x20004`
ADSP-21062	`0x20000`	`0x20004`
ADSP-21065L	`0x8000`	`0x8004`
ADSP-21160	`0x40000`	`0x40004`

1 The reset vector address must not contain a valid instruction since it is not executed during the booting sequence. Place a `NOP` or `IDLE` instruction at this location.

The boot type selection directs the system to prepare the appropriate boot kernel.

Boot Mode Selection

The state of various pins selects the processor boot mode. For the ADSP-21060, ADSP-21061, ADSP-21062, and ADSP-21160 processors, refer to Table 4-3 and Table 4-4. For the ADSP-21065L processors, refer to Table 4-5 and Table 4-6.

Table 4-3. ADSP-21060/061/062 and ADSP-21160 Boot Mode Pins

Pin	Type	Description
EBOOT	I	EPROM boot. When EBOOT is high, the processor boot-loads from an 8-bit EPROM through the processor's external port. When EBOOT is low, the LBOOT and $\overline{\text{BMS}}$ pins determine the booting mode.
LBOOT	I	Link port boot. When LBOOT is high and EBOOT is low, the processor boots from another SHARC through the link port. When LBOOT is low and EBOOT is low, the processor boots from a host processor through the processor's external port.
$\overline{\text{BMS}}$	I/O/T[1]	Boot memory select. When boot-loading from an EPROM (EBOOT=1 and LBOOT=0), this pin is an *output* and serves as the chip select for the EPROM. In a multiprocessor system, $\overline{\text{BMS}}$ is output by the bus master. When host-booting or link-booting (EBOOT=0), $\overline{\text{BMS}}$ is an *input* and must be high.

1 Three-statable in EPROM boot mode (when $\overline{\text{BMS}}$ is an output).

Table 4-4. ADSP-21060/061/062 and ADSP-21160 Boot Modes

EBOOT	LBOOT	BMS	Boot Mode
0	0	0 (Input)	No-boot (processor executes from external memory)
0	0	1 (Input)	Host processor
0	1	0 (Input)	Reserved
0	1	1 (Input)	Link port
1	0	Output	EPROM ($\overline{\text{BMS}}$ is chip select)
1	1	x (Input)	Reserved

Table 4-5. ADSP-21065L Boot Mode Pins

Pin	Type	Description
$\overline{\text{BMS}}$	I/O/T[1]	Boot memory select When BSEL is low, $\overline{\text{BMS}}$ is an input pin and selects between host boot mode and no-boot mode. In no-boot mode, the processor executes from external memory. For no-boot mode, connect $\overline{\text{BMS}}$ to ground. For host boot mode, connect $\overline{\text{BMS}}$ to VDD. When BSEL is high, $\overline{\text{BMS}}$ is an output pin and the processor starts up in EPROM boot mode. Connect $\overline{\text{BMS}}$ to the EPROM's chip select.
BSEL	I	EPROM boot select Hardwire this signal; it is used for system configuration. When BSEL is high, the processor starts up in EPROM boot mode. The processor assumes the EPROM data bus is 8 bits wide. Connect BSEL to the processor data bus in LSB alignment. When BSEL is low, $\overline{\text{BMS}}$ determines the booting mode. Connect BSEL to ground.

1 Three-statable in EPROM boot mode (when $\overline{\text{BMS}}$ is an output).

Table 4-6. ADSP-21065L Boot Modes

BSEL	BMS	Description
0	1	No-boot mode. The processor executes from external memory at location 0x20004.
0	1	Host boot mode. The processor defaults to an 8-bit host bus width.
1	Output	EPROM boot mode. The processor assumes an 8-bit EPROM data bus width. Connect to the data bus in LSB alignment.

ADSP-2106x/21160 Boot Modes

The ADSP-2106x/21160 processors support these boot modes: EPROM, host, and link. The following sections describe each of the modes.

- "EPROM Boot Mode" on page 4-7
- "Host Boot Mode" on page 4-11
- "Link Port Boot Mode" on page 4-15
- "No-Boot Mode" on page 4-16

For multiprocessor booting, refer to "ADSP-2106x/21160 Multi-Application (Multi-DXE) Management" on page 4-23.

EPROM Boot Mode

The ADSP-2106x/21160 processor is configured for EPROM boot through the external port when the EBOOT pin is high and the LBOOT pin is low. These settings cause the $\overline{\text{BMS}}$ pin to become an output, serving as chip select for the EPROM. Table 4-7 lists all PROM-to-processor connections.

Table 4-7. PROM Connections to ADSP-2106x/21160 Processors

Processor	Connection
ADSP-21060/61/62	PROM/EPROM connects to DATA23–16 pins
ADSP-21065L	PROM/EPROM connects to DATA7–0 pins
ADSP-21160	PROM/EPROM connects to DATA39–32 pins
ADSP-21xxx	Address pins of PROM connect to lowest address pins of any processor
ADSP-21xxx	Chip select connects to the $\overline{\text{BMS}}$ pin
ADSP-21060/61/62/65L	Output enable connects to the $\overline{\text{RD}}$ pin
ADSP-21160	Output enable connects to $\overline{\text{RDH}}$ pin

During reset, the ACK line is pulled high internally with a 2K ohm equivalent resistor and is held high with an internal keeper latch. It is not necessary to use an external pull-up resistor on the ACK line during booting or at any other time.

The DMA channel parameter registers are initialized at reset for EPROM booting as shown in Table 4-8 and Table 4-9. The count is initialized to 0x0100 to transfer 256 words to internal memory. The external count register (ECx), which is used when external addresses (BMS space) are generated by the DMA controller, is initialized to 0x0600 (0x100 words at six bytes per word).

Table 4-8. DMA Settings for ADSP-2106x EPROM Booting

DMA Setting		Processor Model	
		ADSP-21060/61/62	ADSP-21065L
BMS space		4M x 8-bit	8M x 8-bit
DMA channel		DMAC6 = 0x2A1	DMAC0 = 0x2A1
II6	IIEP0	0x20000	0x8000
IM6	IMEP0	0x1 (implied)	0x1 (implied)
C6	CEP0	0x100	0x100
EI6	EIEP0	0x80 0000	0x40 0000
EM6	EMEP0	0x1 (implied)	0x1 (implied)
EC6	ECEP0	0x600	0x600
IRQ vector		0x20040	0x8040

Table 4-9. DMA Settings for ADSP-21160 EPROM Booting

DMA Setting	ADSP-21160 Processor
BMS space	8M x 8-bit
DMA channel	DMAC10 = 0x4A1

Table 4-9. DMA Settings for ADSP-21160 EPROM Booting (Cont'd)

DMA Setting	ADSP-21160 Processor
II10	0x40000
IM10	0x1 (implied)
C10	0x100
EI10	0x800000
EM10	0x1 (implied)
EC10	0x600
IRQ vector	0x40050

After the processor's RESET pin goes inactive on start-up, a SHARC system configured for EPROM boot undergoes the following boot-loading sequence:

1. The processor $\overline{\text{BMS}}$ pin becomes the boot EPROM chip select.
2. The processor goes into an idle state, identical to that caused by the IDLE instruction. The program counter (PC) is set to the processor reset vector address (refer to Table 4-2 on page 4-4).
3. The DMA controller reads 8-bit EPROM words, packs them into 48-bit instruction words, and transfers them into internal memory (low-to-high byte packing order) until the 256 words are loaded.
4. The DMA parameter registers for appropriate DMA channels are initialized, as shown in Table 4-8 and Table 4-9. The external port DMA channel (6 or 10) becomes active following reset; it is initialized to set external port DMA enable and selects DTYPE for instruction words. The packing mode bits (PMODE) are ignored, and 48- to 8-bit packing is forced with least significant word first. The UBWS and UBWM fields of the WAIT register are initialized to generate six wait states for the EPROM access in unbanked external memory space.

5. The processor begins 8-bit DMA transfers from the EPROM to internal memory using the following external port data bus lines:

   D23–16 for the ADSP-21060/61/62 processors
   D7–0 for the ADSP-21065L processors
   D39–32 for the ADSP-21160 processors

6. Data transfers begin and increment after each access. The external address lines (ADDR31–0) start at:

   0x40 0000 for the ADSP-21060/61/62 processors
   0x00 0000 for the ADSP-21065L processors
   0x80 0000 for the ADSP-21160 processors

7. The processor $\overline{\text{RD}}$ pin asserts as in a normal memory access, with six wait states (seven cycles).

8. After finishing DMA transfers to load the boot kernel into the processor, the BSO bit is cleared in the SYSCON register, deactivating the $\overline{\text{BMS}}$ pin and activating normal external memory select.

   The boot kernel uses three copies of SYSCON—one that contains the original value of SYSCON, a second that contains SYSCON with the BSO bit set (allowing the processor to gain access to the boot EPROM), and a third with the BSO bit cleared.

   When BSO=1, the EPROM packing mode bits in the DMACx control register are ignored and 8- to 48-bit packing is forced. (8-bit packing is available only during EPROM booting or when BSO is set.) When an external port DMA channel is being used in conjunction with the BSO bit, none of the other three channels may be used. In this mode, $\overline{\text{BMS}}$ is not asserted by a core processor access but only by a DMA transfer. This allows the boot kernel to perform other external accesses to non-boot memory.

The EPROM is automatically selected by the $\overline{\text{BMS}}$ pin after reset, and other memory select pins are disabled. The processor's DMA controller reads the 8-bit EPROM words, packs them into 48-bit instruction words, and transfers them to internal memory until 256 words have been loaded. The master DMA internal and external count registers (Cx and ECx) decrement after each EPROM transfer. When both counters reach zero, DMA transfer has stopped and RTI returns the program counter to the address where the kernel starts.

To EPROM boot a single-processor system, include the executable on the command-line without a switch. Do not use the -id#exe switch with ID=0 (see "ADSP-2106x/21160 Processor ID Numbers" on page 4-24).

The WAIT register UBWM (used for EPROM booting) is initialized at reset to both internal wait and external acknowledge required. The internal keeper latch on the ACK pin initially holds acknowledge high (asserted). If acknowledge is driven low by another device during an EPROM boot, the keeper latch may latch acknowledge low.

The processor views the deasserted (low) acknowledge as a hold off from the EPROM. In this condition, wait states are continually inserted, preventing completion of the EPROM boot. When writing a custom boot kernel, change the WAIT register early within the boot kernel so UBWM is set to internal wait mode (01).

Host Boot Mode

The ADSP-2106x/21160 processors accept data from a 8- and 16-bit host microprocessor (or other external device) through the external port EPB0 and pack boot data into 48-bit instructions using an appropriate DMA channel. The host is selected when the EBOOT and LBOOT inputs are low and $\overline{\text{BMS}}$ is high. Configured for host booting, the processor enters the slave mode after reset and waits for the host to download the boot program. Table 4-10 lists host connections to processors.

Table 4-10. Host Connections to ADSP-2106x/21160 Processors

Processor	Connection/Data Bus Pins
ADSP-21060/61/62	Host connected to DATA47–16 or DATA31–16 pins (based on HPM bits)
ADSP-21065L	Host connected to DATA31–0 or DATA15–0 or DATA7–0 pins (based on HBW bits)
ADSP-21160	Host connected to DATA63–32 or DATA47–31 pins (based on HPM bits)
ADSP-21060/61/62/65L	ADSP-21065L host address to IOP registers only
ADSP-21160	ADSP-21160 host address to IOP registers and internal memory

After reset, the processor goes into an idle stage with:

- PC set to address 0x20004 on the ADSP-21060/61/62 processors
- PC set to address 0x8004 on the ADSP-21065L processors
- PC set to address 0x40004 on the ADSP-21160 processors

The parameter registers for the external port DMA channel (0, 6, or 10) are initialized as shown in Table 4-8 and Table 4-9, except that registers EIx, EMx and ECx are not initialized and no DMA transfers start.

The DMA channel control register (DMAC6) for the ADSP-21060/61/62 processors, DMAC0 for the ADSP-21065L processors, or DMAC10 for the ADSP-21160 processors) is initialized, which allows external port DMA enable and selects DTYPE for instruction words, PMODE for 16- to 48-bit word packing (8- to 48-bit for the ADSP-21065L processors), and least significant word first.

Because the host processor is accessing the EPB0 external port buffer, the HPM host packing mode bits of the SYSCON register must correspond to the external bus width specified by the PMODE bits of DMACx control register.

For a different packing mode, the host must write to DMACx and SYSCON to change the PMODE and HBW (HPW for the ADSP-21065L processors) setting. The host boot file created by the loader utility requires the host processor to perform the following sequence of actions:

1. The host initiates the synchronous booting operation (synchronous not valid for the ADSP-21065L processors) by asserting the processor $\overline{\text{HBR}}$ input pin, informing the processor that the default 8-/16-bit bus width is used. The host may optionally assert the $\overline{\text{CS}}$ chip select input to allow asynchronous transfers.

2. After the host receives the $\overline{\text{HBG}}$ signal (and ACK for synchronous operation or READY for asynchronous operation) from the processor, the host can start downloading instructions by writing directly to the external port DMA buffer 0 or the host can change the reset initialization conditions of the processor by writing to any of the IOP control registers. The host must use data bus pins as shown in Table 4-10.

3. The host continues to write 16-bit words (8-bit for the ADSP-21065L) to EPB0 until the entire program is boot-loaded. The host must wait between each host write to external port DMA buffer 0.

After the host boot-loads the first 256 instructions of the boot kernel, the initial DMA transfers stop, and the boot kernel:

1. Activates external port DMA channel interrupt (EPOI), stores the DMACx control setting in R2 for later restore, clears DMACx for new setting, and sets the BUSLCK bit in the MODE2 register to lock out the host.

2. Stores the SYSCON register value in R12 for restore.

3. Enables interrupts and nesting for DMA transfer, sets up the IMASK register to allow DMA interrupts, and sets up the MODE1 register to enable interrupts and allow nesting.

4. Loads the DMA control register with 0x00A1 and sets up its parameters to read the data word by word from external buffer 0.

   Each word is read into the reset vector address (refer to Table 4-2 on page 4-4) for dispatching. The data through this buffer has a structure of boot section which could include more than one initialization block.

5. Clears the BUSLCK bit in the MODE2 register to let the host write in the external buffer 0 right after the appropriate DMA channel is activated.

   For information on the data structure of the boot section and initialization, see "ADSP-2106x/21160 Processor Boot Steams" on page 4-17.

6. Loads the first 256 words of target the executable file during the final initialization stage, and then the kernel overwrites itself.

The final initialization works the same way as with EPROM booting, except that the BUSLCK bit in the MODE2 register is cleared to allow the host to write to the external port buffer.

The default boot kernel for host booting assumes IMDW is set to 0 during boot-loading, except during the final initialization stage. When using any power-up booting mode, the reset vector address (refer to Table 4-2 on page 4-4) must not contain a valid instruction because it is not executed during the booting sequence. Place a NOP or IDLE instruction at this location.

If the boot kernel initializes external memory, create a custom boot kernel that sets appropriate values in the SYSCON and WAIT register. Be aware that the value in the DMA channel register is non-zero, and IMASK is set to allow DMA channel register interrupts. Because the DMA interrupt remains enabled in IMASK, this interrupt must be cleared before using the DMA channel again. Otherwise, unintended interrupts may occur.

A master SHARC processor may boot a slave SHARC processor by writing to its DMACx control register and setting the packing mode (PMODE) to 00. This allows instructions to be downloaded directly without packing. The wait state setting of 6 on the slave processor does not affect the speed of the download since wait states affect bus master operation only.

Link Port Boot Mode

Link port boot is supported on all SHARC processors except the ADSP-21061 and ADSP-21065L processors.

When link-boot the ADSP-2106x/21160 SHARC processors, the processor receives data from 4-bit link buffer 4 and packs boot data into 48-bit instructions using the appropriate DMA channels (DMA channel 6 for the ADSP-2106x processors, DMA channel 8 for the ADSP-21160 processors).

Link port mode is selected when the EBOOT is low and LBOOT and $\overline{\text{BMS}}$ are high. The external device must provide a clock signal to the link port assigned to link buffer 4. The clock can be any frequency, up to a maximum of the processor clock frequency. The clock falling edges strobe the data into the link port. The most significant 4-bit nibble of the 48-bit instruction must be downloaded first. The link port acknowledge signal generated by the processor can be ignored during booting since the link port cannot be preempted by another DMA channel.

Link booting is similar to host booting—the parameter registers (IIx and Cx) for DMA channels are initialized to the same values. The DMA channel 6 control register (DMAC6) is initialized to 0x00A0, and the DMA channel 10 control register (DMAC10) is initialized to 0x100000. This disables external port DMA and selects DTYPE for instruction words. The LCTL and LCOM link port control registers are overridden during link booting to allow link buffer 4 to receive 48-bit data.

After booting completes, the IMASK remains set, allowing DMA channel interrupts. This interrupt must be cleared before link buffer 4 is again enabled; otherwise, unintended link interrupts may occur.

No-Boot Mode

No-boot mode causes the processor to start fetching and executing instructions at address 0x400004 (ADSP-2106x), 0x20004 (ADSP-21065L), and 0x800004 (ADSP-21160) in external memory space. All DMA control and parameter registers are set to their default initialization values. The loader utility is not intended to support no-boot mode.

ADSP-2106x/21160 Boot Kernels

The boot-loading process starts with a transfer of the boot kernel program into the processor memory. The boot kernel sets up the processor and loads boot data. After the boot kernel finishes initializing the rest of the system, the boot kernel loads boot data over itself with a final DMA transfer.

Boot kernels are loaded at reset into program segment seg_ldr, which is defined in:

- 06x_ldr.ldf for the ADSP-2106x processors
- 065L_ldr.ldf for the ADSP-21065L processors
- 160_ldr.ldf for the ADSP-21160 processors

The files are stored in the <*install_path*>\21k\ldr (ADSP-2106x processors) and \211xx\ldr (ADSP-21160 processors) directories of VisualDSP++.

The default boot kernel files shipped with VisualDSP++ are listed in Table 4-11.

Table 4-11. ADSP-2106x/21160 Default Boot Kernel Files

Processor	PROM Booting	Link Booting	Host Booting
ADSP-21060	060_prom.asm	060_link.asm	060_host.asm
ADSP-21065L	065L_prom.asm	N/A	065L_host.asm
ADSP-21160	160_prom.asm	160_link.asm	160_host.asm

Once the boot kernel has been loaded successfully into the processor, the kernel follows the following sequence:

1. Each boot kernel begins with general initializations for the DAG registers, appropriate interrupts, processor ID information, and various SDRAM or WAIT state initializations.
2. Once the boot kernel has finished the task of initializing the processor, the kernel initializes processor memory, both internal and external, with user application code.

ADSP-2106x/21160 Processor Boot Steams

The structure of a loader file enables the boot kernel to load code and data, block by block. In the loader file, each block of code or data is preceded by a block header, which describes the block —length, placement, and data or instruction type. After the block header, the loader utility outputs the block body, which includes the actual data or instructions for placement in the processor memory. The loader utility, however, does not output a block body if the actual data or instructions are all zeros in value. This type of block called a zero block. Table 4-12 describes the block header and block body formats.

The loader utility identifies the data type in the block header with a 16-bit tag that precedes the block. Each type of initialization has a unique tag number. The tag numbers and block types are shown in Table 4-13.

Table 4-12. Boot Block Format

Block header	First word	Bits 16–47 are not used Bits 0–15 define the type of data block (tag)
	Second word	Bits 16–47 are the start address of the block Bits 0–15 are the word count for the block
Block body (if not a zero block)		Word 1 (48 bits) Word 2 (48 bits)

Table 4-13. ADSP-2106x/21160 Processor Loader Block Tags

Tag Number	Block Type	Tag Number	Block Type
0x0000	final init	0x000A	zero pm48
0x0001	zero dm16	0x000B	init pm16
0x0002	zero dm32	0x000C	init pm32
0x0003	zero dm40	0x000E	init pm48
0x0004	init dm16	0x000F	zero dm64 (ADSP-21160 only)
0x0005	init dm32	0x0010	init dm64 (ADSP-21160 only)
0x0007	zero pm16	0x0011	zero pm64 (ADSP-21160 only)
0x0008	zero pm32	0x0012	init pm64 (ADSP-21160 only)
0x0009	zero pm40		

The kernel enables the boot port (external or link) to read the block header. After reading information from the block header, the kernel places the body of the block in the appropriate place in memory if the block has a block body, or initializes in the appropriate place with zero values in the memory if the block is a zero block.

The final section, which is identified by a tag of 0x0, is called the final initialization section. This section has self-modifying code that, when executed, facilitates a DMA over the kernel, replacing it with user applica-

tion code that actually belongs in that space at run time. The final initialization code also takes care of interrupts and returns the processor registers, such as SYSCON and DMAC or LCTL, to their default values.

When the loader utility detects the final initialization tag, it reads the next 48-bit word. This word indicates the instruction to load into the 48-bit PX register after the boot kernel finishes initializing memory.

The boot kernel requires that the interrupt, external port (or link port address, depending on the boot mode) contains an RTI instruction. This RTI is inserted automatically by the loader utility to guarantee that the kernel executes from the reset vector, once the DMA that overwrites the kernel is complete. A last remnant of the kernel code is left at the reset vector location to replace the RTI with the user's intended code. Because of this last kernel remnant, user application code should not use the first location of the reset vector. This first location should be a NOP or IDLE instruction. The kernel automatically completes, and the program controller begins sequencing the user application code at the second location in the processor reset vector space.

When the boot process is complete, the processor automatically executes the user application code. The only remaining evidence of the boot kernel is at the first location of the interrupt vector. Almost no memory is sacrificed to the boot code.

Boot Kernel Modification and Loader Issues

Some systems require boot kernel customization. The operation of other tools (such as the C/C++ compiler) is influenced by whether the boot kernel is used.

When producing a boot-loadable file, the loader utility reads a processor executable file and uses information in it to initialize the memory. However, the loader utility cannot determine how the processor SYSCON and WAIT registers are to be configured for external memory loading in the system.

If you modify the boot kernel by inserting values for your system, you must rebuild it before generating the boot-loadable file. The boot kernel contains default values for `SYSCON`. The initialization code can be found in the comments in the boot kernel source file.

After modifying the boot kernel source file, rebuild the boot kernel (`.dxe`) file. Do this from the VisualDSP++ IDDE (refer to VisualDSP++ online Help for details), or rebuild the boot kernel file from the command line.

When using VisualDSP++, specify the name of the modified kernel executable in the **Kernel file** box on the **Kernel** page of the **Project Options** dialog box.

If you modify the boot kernel for EPROM, host, or link boot modes, ensure that the `seg_ldr` memory segment is defined in the `.ldf` file. Refer to the source of the segment in the `.ldf` file located in the `<`*`install_path`*`>\21k\ldr\` or `<`*`install_path`*`>\211xx\ldr\` directory of VisualDSP++.

The loader utility generates a warning when vector address (`0x20004` for the ADSP-21060/61/62 processors, `0x40004` for the ADSP-21160 processors, or `0x8004` for the ADSP-21065L processors) does not contain `NOP` or `IDLE`. Because the boot kernel uses this address for the first location of the reset vector during the boot-load process, avoid placing code at this address. When using any of the processor's power-up boot modes, ensure that the address does not contain a critical instruction. Because the address is not executed during the booting sequence, place a `NOP` or `IDLE` instruction at this location.

The boot kernel project can be rebuilt from the VisualDSP++ IDDE. The command-line can also be used to rebuild various default boot kernels for the ADSP-2106x/21160 processors.

EPROM Booting. The default boot kernel source file for the ADSP-2106x EPROM booting is `060_prom.asm`. Copy this file to `my_prom.asm` and modify it to suit your system. Then use the following commands to rebuild the boot kernel.

```
easm21k -21060 my_prom.asm
```

or

```
easm21k -proc ADSP-21060 my_prom.asm
linker -T 060_ldr.ldf my_prom.doj
```

Host Booting. The default boot kernel source file for the ADSP-2106x host booting is `060_host.asm`. Copy this file to `my_host.asm` and modify it to suit your system. Then use the following commands to rebuild the boot kernel.

```
easm21k -21060 my_host.asm
```

or

```
easm21k -proc ADSP-21060 my_host.asm
linker -T 060_ldr.ldf my_host.doj
```

Link Port Booting. The default boot kernel source file for the ADSP-2106x link port booting is `060_link.asm`. Copy this file to `my_link.asm` and modify it to suit your system. Then use the following commands to rebuild the boot kernel:

```
easm21k -21060 my_link.asm
```

or

```
easm21k -proc ADSP-21060 my_link.asm
linker -T 060_ldr.ldf my_link.doj
```

Rebuilding Boot Kernels

To rebuild the PROM boot kernel for the ADSP-21065L processors, use these commands:

```
easm21k -21065L my_prom.asm
```

or

```
easm21k -proc ADSP-21065L my_prom.asm
linker -T 065L_ldr.ldf my_prom.doj
```

To rebuild the PROM boot kernel for the ADSP-21160 processors, use these commands.

```
easm21k -21160 my_prom.asm
```

or

```
easm21k -proc ADSP-21160 my_prom.asm
linker -T 160_ldr.ldf my_prom.doj
```

ADSP-2106x/21160 Interrupt Vector Table

If an ADSP-2106x/21160 SHARC processor is booted from an external source (EPROM, host, or another SHARC processor), the interrupt vector table is located in internal memory. If, however, the processor is not booted and executes from external memory, the vector table must be located in external memory.

The `IIVT` bit of the `SYSCON` control register can be used to override the boot mode in determining where the interrupt vector table is located. If the processor is not booted (no-boot mode), setting `IIVT` to `1` selects an internal vector table, and setting `IIVT` to `0` selects an external vector table. If the processor is booted from an external source (any mode other than no-boot mode), `IIVT` has no effect. The `IIVT` default initialization value is `0`.

Refer to *EE-56: Tips & Tricks on the ADSP-2106x EPROM and HOST bootloader*, *EE-189: Link Port Tips and Tricks for ADSP-2106x and ADSP-2116x*, and *EE-77: SHARC Link Port Booting* on the Analog Devices Web site for more information.

ADSP-2106x/21160 Multi-Application (Multi-DXE) Management

Currently, the loader utility generates single-processor loader files for host and link port boot modes. As a result, the loader utility supports multiprocessor EPROM boot mode only. The application code must be modified for a multiprocessor system boot in host and link port modes.

The loader utility can produce boot-loadable files that permit the ADSP-2106x/21160 SHARC processors in a multiprocessor system to boot from a single EPROM. In such a system, the $\overline{\texttt{BMS}}$ signals from each SHARC processor are `OR`'ed together to drive the chip select pin of the EPROM. Each processor boots in turn, according to its priority. When the last processor finishes booting, it must inform the processors to begin program execution.

Besides taking turns when booting, EPROM boot of multiple processors is similar to a single-processor EPROM boot.

When booting a multiprocessor system through a single EPROM:

- Connect all $\overline{\texttt{BMS}}$ pins to EPROM.
- Processor with ID# of `1` boots first. The other processors follow.
- The EPROM boot kernel accepts multiple `.dxe` files and reads the `ID` field in `SYSTAT` to determine which area of EPROM to read.
- All processors require a software flag or hardware signal (`FLAG` pins) to indicate that booting is complete.

When booting a multiprocessor system through an external port:

- The host can use the host interface.
- A SHARC processor that is EPROM-, host-, or link-booted can boot the other processors through the external port (host boot mode).

For multiprocessor EPROM booting, select the **Multiprocessor** check box on the **Load** page of the **Project Options** dialog box or specify the `-id1exe=` switch on the loader command line. These options specify the executable file targeted for a specific processor.

Do not use the `-id1exe=` switch to EPROM-boot a single processor whose `ID` is `0`. Instead, name the executable file on the command line without a switch. For a single processor with `ID=1`, use the `-id1exe=` switch.

ADSP-2106x/21160 Processor ID Numbers

A single-processor system requires only one input (`.dxe`) file without any prefix and suffix to the input file name, for example:

```
elfloader -proc ADSP-21060 -bprom Input.dxe
```

A multiprocessor system requires a distinct processor ID number for each input file on the command line. A processor ID is provided via the `-id#exe=`*`filename`*`.dxe` switch, where `#` is `0` to `6`.

In the following example, the loader utility processes the input file `Input1.dxe` for the processor with an ID of `1` and the input file `Input2.dxe` for the processor with an ID of `2`.

```
elfloader -proc ADSP-21060 -bprom -id1exe=Input1.dxe
          -id2exe=Input2.dxe
```

If the executable for the `#` processor is identical to the executable of the `N` processor, the output loader file contains only one copy of the code from the input file.

```
elfloader -proc ADSP-21060 -bprom -id1exe=Input.dxe -id2ref=1
```

The loader utility points the `id(2)exe` loader jump table entry to the `id(1)exe` image, effectively reducing the size of the loader file.

ADSP-2106x/21160 Processor Loader Guide

Loader operations depend on the loader options, which control how the loader utility processes executable files. You select features such as boot modes, boot kernels, and output file formats via the loader options. These options are specified on the loader utility's command line or via the **Load** page of the **Project Options** dialog box in the VisualDSP++ environment. When you open the **Load** page, the default loader settings for the selected processor are already set. Use the **Additional Options** box to enter options that have no dialog box equivalent.

Option settings on the **Load** page correspond to switches displayed on the command line.

For detailed information about the ADSP-2106x/21160 processor loader property page, refer to the VisualDSP++ online help.

These sections describe how to produce a bootable loader (`.ldr`) file:

- "Using ADSP-2106x/21160 Loader Command Line" on page 4-26
- "Using VisualDSP++ Interface (Load Page)" on page 4-32

Using ADSP-2106x/21160 Loader Command Line

Use the following syntax for the SHARC loader command line.

```
elfloader inputfile -proc part_number -switch [-switch …]
```

where:

- *inputfile*—Name of the executable (`.dxe`) file to be processed into a single boot-loadable file. An input file name can include the drive and directory. Enclose long file names within straight quotes, "`long file name`".
- `-proc` *part_number*—Part number of the processor (for example, `-proc ADSP-21062`) for which the loadable file is built. The `-proc` switch is mandatory.
- `-switch` …—One or more optional switches to process. Switches select operations and boot modes for the loader utility. A list of all switches and their descriptions appear in Table 4-15 on page 4-28.

Command-line switches are not case-sensitive and placed on the command line in any order.

The following command line,

```
elfloader p0.dxe -bprom -fhex -l 060_prom.dxe -proc ADSP-21060
```

runs the loader utility with:

- `p0.dxe`—Identifies the executable file to process into a boot-loadable file. The absence of the `-o` switch causes the output file name to default to `p0.ldr`.
- `-bprom` —Specifies EPROM booting as the boot type for the boot-loadable file.
- `-fhex` —Specifies Intel hex-32 format for the boot-loadable file.

- `-l 060_prom.exe`—Specifies `060_prom.exe` as the boot kernel file to be used in the boot-loadable file.
- `-proc ADSP-21060`—Identifies the processor model as ADSP-21060.

File Searches

File searches are important in loader processing. The loader utility supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur as described on page 1-17.

File Extensions

Some loader switches take a file name as an optional parameter. Table 4-14 lists the expected file types, names, and extensions.

Table 4-14. File Extensions

Extension	File Description
`.dxe`	Input executable files and boot kernel files. The loader utility recognizes overlay memory files (`.ovl`) and shared memory files (`.sm`), but does not expect these files on the command line. Place `.ovl` and `.sm` files in the same directory as the `.dxe` file that refers to them. The loader utility finds the files when processing the `.dxe` file. The `.ovl` and `.sm` files may also be placed in the `.ovl` and `.sm` file output directory specified in the `.ldf` file or current working directory.
`.ldr`	Loader output file

ADSP-2106x/21160 Loader Command-Line Switches

Table 4-15 is a summary of the ADSP-2106x and ADSP-21160 loader switches.

Table 4-15. ADSP-2106x/21160 Loader Command-Line Switches

Switch	Description

`-bprom`
`-bhost`
`-blink`
`-bJTAG`
	Specifies the boot mode. The `-b` switch directs the loader utility to prepare a boot-loadable file for the specified boot mode. Valid boot modes include PROM, host, and link.
	For the ADSP-21020 processors, JTAG is the only permitted boot mode.
	If `-b` does not appear on the command line, the default is `-bprom`.
	To use a custom boot kernel, the boot type selected with the `-b` switch must correspond to the boot kernel selected with the `-l` switch. Otherwise, the loader utility automatically selects a default boot kernel based on the selected boot type (see "ADSP-2106x/21160 Boot Kernels" on page 4-16).

`-caddress`	Custom option. This switch directs the loader utility to use the specified address. Valid addresses are: • `20004` and `20040` for the ADSP-2106x processors • `8004` and `8040` for the ADSP-21065L processors • `40000` and `40050` for the ADSP-21160 processors The loader utility obtains the proper address even when this switch is absent from the command line.
`-e filename`	Except shared memory. The `-e` switch omits the specified shared memory (`.sm`) file from the output loader file. Use this option to omit the shared parts of the executable file intended to boot a multiprocessor system. To omit multiple `.sm` files, repeat the switch and parameter multiple times on the command line. For example, to omit two files, use: `-e fileA.sm -e fileB.sm`. In most cases, it is not necessary to use the `-e` switch: the loader utility processes the `.sm` files efficiently—includes a single copy of the code and data from each `.sm` file in a loader file.

Table 4-15. ADSP-2106x/21160 Loader Command-Line Switches

Switch	Description

`-fhex`
`-fASCII`
`-fbinary`
`-finclude`
`-fS1`
`-fS2`
`-fS3`
	Specifies the format of the boot-loadable file (Intel hex-32, ASCII, S1, S2, S3, binary, or include). If the `-f` switch does not appear on the command line, the default boot file format is Intel hex-32 for PROM, and ASCII for host or link.
	Available formats depend on the boot type selection (`-b` switch):
	• For PROM boot type, select a hex, ASCII, S1, S2, S3, or include format.
	• For host or link boot type, select an ASCII, binary, or include format.

`-h`
or
`-help`
	Command-line help. Outputs a list of the command-line switches to standard out and exits. Type `elfloader -proc ADSP-21xxx -h`, where `xxx` is `060`, `061`, `062`, `065L`, or `160` to obtain help for SHARC processors. By default, the `-h` switch alone provides help for the loader driver.

`-id#exe=`*`filename`*	Specifies the processor ID. The `-id#exe=` switch directs the loader utility to use the processor ID (*#*) for the corresponding executable file (*`filename`* parameter) when producing a boot-loadable file for a multiprocessor system. This switch is used to produce a boot-loadable file that boots multiple processors from a single EPROM. Valid values for *#* are `1`, `2`, `3`, `4`, `5`, and `6`. Do not use this switch for single-processor systems. For single-processor systems, use *`filename`* as a parameter without a switch. For more information, refer to "ADSP-2106x/21160 Processor ID Numbers" on page 4-24.
`-id#ref=`*`N`*	Points the processor ID (#) loader jump table entry to the ID (*`N`*) image. If the executable file for the (*#*) processor is identical to the executable of the (*`N`*) processor, the switch can be used to set the PROM start address of the processor with ID of # to be the same as for the processor with ID of *`N`*. This effectively reduces the size of the loader file by providing a single copy of an executable to two or more processors in a multiprocessor system. For more information, refer to "ADSP-2106x/21160 Processor ID Numbers" on page 4-24.

Table 4-15. ADSP-2106x/21160 Loader Command-Line Switches

Switch	Description
`-l kernelfile`	Directs the loader utility to use the specified *`kernelfile`* as the boot-loading routine in the output boot-loadable file. The boot kernel selected with this switch must correspond to the boot type selected with the `-b` switch. If the `-l` switch does not appear on the command line, the loader searches for a default boot kernel file. Based on the boot type (`-b` switch), the loader utility searches in the processor-specific loader directory for the boot kernel file as described in "ADSP-2106x/21160 Boot Kernels" on page 4-16.
`-o filename`	Directs the loader utility to use the specified *`filename`* as the name for the loader output file. If not specified, the default name is *`input-file`*`.ldr`.
`-paddress`	PROM start address. Places the boot-loadable file at the specified address in the EPROM. If the `-p` switch does not appear on the command line, the loader utility starts the EPROM file at address `0x0`; this EPROM address corresponds to `0x800000` on the ADSP-21060/21061/21062, ADSP-21065L, and ADSP-21160 processors.
`-proc processor`	Specifies the processor. This a mandatory switch. The *`processor`* is one of the following: `ADSP-21060`, `ADSP-21061`, `ADSP-21062`, `ADSP-21065L`, `ADSP-21160`

Table 4-15. ADSP-2106x/21160 Loader Command-Line Switches

Switch	Description
`-si-revision #|none|any`	The `-si-revision {#|none|any}` switch provides a silicon revision of the specified processor. The switch parameter represents a silicon revision of the processor specified by the `-proc` *`processor`* switch. The parameter takes one of three forms: • The `none` value indicates that the VisualDSP++ ignores silicon errata. • The # value indicates one or more decimal digits, followed by a point, followed by one or two decimal digits. Examples of revisions are: `0.0`; `1.12`; `23.1`. Revision `0.1` is distinct from and "lower" than revision `0.10`. The digits to the left of the point specify the chip tapeout number; the digits to the right of the point identify the metal mask revision number. The number to the right of the point cannot exceed decimal `255`. • The `any` value indicates that VisualDSP++ produces an output file that can be run at any silicon revision. The switch generates either a warning about any potential anomalous conditions or an error if any anomalous conditions occur. In the absence of the switch parameter (a valid revision value)—`-si-revision` alone or with an invalid value—the loader utility generates an error.
`-t#`	(Host boot only) Specifies timeout cycles; for example, `-t100`. Limits the number of cycles that the processor spends initializing external memory with zeros. Valid timeout values (#) range from `3` to `32765` cycles; `32765` is the default. The # is directly related to the number of cycles the processor locks the bus for boot-loading, instructing the processor to lock the bus for no more than two times the timeout number of cycles. When working with a fast host that cannot tolerate being locked out of the bus, use a relatively small timeout value.
`-use32bitTagsfor ExternalMemory-Blocks`	Directs the loader utility to treat the external memory sections as 32-bit sections, as specified in the `.ldf` file and does not pack them into 48-bit sections before processing. This option is useful if the external memory sections are packed by the linker and do not need the loader utility to pack them again.

Table 4-15. ADSP-2106x/21160 Loader Command-Line Switches

Switch	Description
`-v`	Outputs verbose loader utility messages and status information as the the utility processes files.
`-version`	Directs the loader utility to show its version information. Type `elfloader -version` to display the version of the loader drive. Add the `-proc` switch, for example, `elfloader -proc ADSP-21062 -version` to display version information of both loader drive and SHARC loader utility.

Using VisualDSP++ Interface (Load Page)

After selecting a **Loader file** as the target type on the **Project** page in VisualDSP++ **Project Options** dialog box, modify the default options on the **Load: Processor** page (also called loader property page). Click **OK** to save the selections. Selecting **Build Project** from the **Project** menu generates a loader file. For information relative to a specific processor, refer to the VisualDSP++ online help for that processor.

VisualDSP++ invokes the `elfloader` utility to build the output file. The **Load** page buttons and fields correspond to loader command-line switches and parameters (see Table 4-15 on page 4-28). Use the **Additional Options** box to enter options that do not have dialog box equivalents.

For the ADSP-21020 processors, the only permitted boot mode is JTAG: **-bJTAG** is automatically entered in the **Additional Options** box.

5 LOADER FOR ADSP-21161 SHARC PROCESSORS

This chapter explains how the loader utility (`elfloader.exe`) is used to convert executable (`.dxe`) files into boot-loadable files for the ADSP-21161 SHARC processors.

Refer to "Introduction" on page 1-1 for the loader utility overview; the introductory material applies to all processor families. Refer to "Loader for ADSP-2106x/21160 SHARC Processors" on page 4-1 for information about the ADSP-21060, ADSP-21061, ADSP-21062, ADSP-21065L, and ADSP-21160 processors. Refer to "Loader for ADSP-2126x/2136x/2137x/2146x SHARC Processors" on page 6-1 for information about the ADSP-2126x and ADSP-2136x processors.

Loader operations specific to the ADSP-21161 SHARC processors are detailed in the following sections.

- "ADSP-21161 Processor Booting" on page 5-2
  Provides general information about various boot modes, including information about boot kernels.
- "ADSP-21161 Processor Loader Guide" on page 5-24
  Provides reference information about the loader utility's graphical user interface, command-line syntax, and switches.

Refer to *EE-177 SHARC SPI Booting*, *EE-199 Link Port Booting on the ADSP-21161 SHARC DSP*, *EE-209 Asynchronous Host Interface on the ADSP-21161 SHARC DSP* on the Analog Devices Processor Web site for related information.

ADSP-21161 Processor Booting

The ADSP-21161 processors support five boot modes: EPROM, host, link port, SPI port, and no-boot (see Table 5-1 and Table 5-2 on page 5-4.) Boot-loadable files for these modes pack boot data into words of appropriate widths and use an appropriate DMA channel of the processor's DMA controller to boot-load the words.

- When booting from an EPROM through the external port, the ADSP-21161 processor reads boot data from an 8-bit external EPROM.
- When booting from a host processor through the external port, the ADSP-21161 processor accepts boot data from 8- or 16-bit host microprocessor.
- When booting through the link port, the ADSP-21161 processor receives boot data through the link port as 4-bit wide data in link buffer 4.
- When booting through the SPI port, the ADSP-21161 processor uses DMA channel 8 of the IO processor to transfer instructions to internal memory. In this boot mode, the processor receives data in the `SPIRx` register.
- In no-boot mode, the ADSP-21161 processors begin executing instructions from external memory.

Software developers who use the loader utility should be familiar with the following operations:

- "Power-Up Booting Process" on page 5-3
- "Boot Mode Selection" on page 5-4
- "ADSP-21161 Processor Boot Modes" on page 5-5
- "ADSP-21161 Processor Boot Kernels" on page 5-16

- "Boot Kernel Modification and Loader Issues" on page 5-18
- "ADSP-21161 Processor Interrupt Vector Table" on page 5-21
- "ADSP-21161 Multi-Application (Multi-DXE) Management" on page 5-21

Power-Up Booting Process

The ADSP-21161 processors include a hardware feature that boot-loads a small, 256-instruction program into the processor's internal memory after power-up or after the chip reset. These instructions come from a program called boot kernel. When executed, the boot kernel facilitates booting of user application code. The combination of the boot kernel and application code comprises the boot-loadable (`.ldr`) file.

At power-up, after the chip reset, the booting process includes the following steps.

1. Based on the boot mode, an appropriate DMA channel is automatically configured for a 256-instruction transfer. This transfer boot-loads the boot kernel program into the processor memory.
2. The boot kernel runs and loads the application executable code and data.
3. The boot kernel overwrites itself with the first 256 words of the application at the end of the booting process. After that, the application executable code starts running.

The boot mode selection directs the system to prepare the appropriate boot kernel.

Boot Mode Selection

The state of the LBOOT, EBOOT, and $\overline{\text{BMS}}$ pins selects the ADSP-21161 processor's boot mode. Table 5-1 and Table 5-2 show how the pin states correspond to the modes.

Table 5-1. ADSP-21161 Boot Mode Pins

Pin	Type	Description
EBOOT	I	EPROM boot – when EBOOT is high, the processor boot-loads from an 8-bit EPROM through the processor's external port. When EBOOT is low, the LBOOT and $\overline{\text{BMS}}$ pins determine booting mode.
LBOOT	I	Link port boot – when LBOOT is high and EBOOT is low, the processor boots from another SHARC processor through the processor's link port. When LBOOT is low and EBOOT is low, the processor boots from a host processor through the processor's external port.
$\overline{\text{BMS}}$	I/O/T[1]	Boot memory select – when boot-loading from EPROM (EBOOT=1 and LBOOT=0), the pin is an *output* and serves as the chip select for the EPROM. In a multiprocessor system, $\overline{\text{BMS}}$ is output by the bus master. When host-booting, link-booting, or SPI-booting (EBOOT=0), $\overline{\text{BMS}}$ is an input and must be high.

1 Three-statable in EPROM boot mode (when $\overline{\text{BMS}}$ is an output).

Table 5-2. ADSP-21161 Boot Mode Pin States

EBOOT	LBOOT	BMS	Booting Mode
1	0	Output	EPROM (connects $\overline{\text{BMS}}$ to EPROM chip select)
0	0	1 (Input)	Host processor
0	1	1 (Input)	Link port
0	1	0 (Input)	Serial port (SPI)
0	0	0 (Input)	No-boot (processor executes from external memory)

ADSP-21161 Processor Boot Modes

The ADSP-21161 processors support these boot modes: EPROM, host, link, and SPI. The following section describe each of the modes.

- "EPROM Boot Mode" on page 5-5
- "Host Boot Mode" on page 5-9
- "Link Port Boot Mode" on page 5-12
- "SPI Port Boot Mode" on page 5-14
- "No-Boot Mode" on page 5-16

For multiprocessor booting, refer to "ADSP-21161 Multi-Application (Multi-DXE) Management" on page 5-21.

EPROM Boot Mode

EPROM boot via the external port is selected when the EBOOT input is high and the LBOOT input is low. These settings cause the $\overline{\text{BMS}}$ pin to become an output, serving as chip select for the EPROM.

The DMAC10 control register is initialized for booting packing boot data into 48-bit instructions. EPROM boot mode uses channel 10 of the IO processor's DMA controller to transfer the instructions to internal memory. For EPROM booting, the processor reads data from an 8-bit external EPROM.

After the boot process loads 256 words into memory locations 0x40000 through 0x400FF, the processor begins to execute instructions. Because most processor programs require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the application. VisualDSP++ includes loading routines (boot kernels) that can load entire programs; see "ADSP-21161 Processor Boot Kernels" on page 5-16 for more information.

Refer to the *ADSP-21161 SHARC DSP Hardware Reference* for detailed information on DMA and system configurations.

Be aware that DMA channel differences between the ADSP-21161 and previous SHARC processors (ADSP-2106x) account for boot differences. Even with these differences, the ADSP-21161 processor supports the same boot capability and configuration as the ADSP-2106x processors. The DMACx register default values differ because the ADSP-21161 processor has additional parameters and different DMA channel assignments. EPROM boot mode uses EPB0, DMA channel 10. Similar to the ADSP-2106x processors, the ADSP-21161 processor boots from DATA23–16.

The processor determines the booting mode at reset from the EBOOT, LBOOT, and $\overline{\text{BMS}}$ pin inputs. When EBOOT=1 and LBOOT=0, the processor boots from an EPROM through the external port and uses $\overline{\text{BMS}}$ as the memory select output. For information on boot mode selection, see the boot memory select pin descriptions in Table 5-1 and Table 5-2 on page 5-4.

When using any of the power-up boot modes, address 0x40004 should not contain a valid instruction since it is not executed during the booting sequence. Place a NOP or IDLE instruction at this location.

EPROM boot (boot space 8M x 8-bit) through the external port requires that an 8-bit wide boot EPROM be connected to the processor data bus pins 23–16 (DATA23–16). The processor's lowest address pins should be connected to the EPROM address lines. The EPROM's chip select should be connected to $\overline{\text{BMS}}$, and its output enable should be connected to $\overline{\text{RD}}$.

In a multiprocessor system, the $\overline{\text{BMS}}$ output is driven by the ADSP-21161 processor bus master only. This allows the wired OR of multiple $\overline{\text{BMS}}$ signals for a single common boot EPROM.

Systems can boot up to six ADSP-21161 processors from a single EPROM using the same code for each processor or differing code for each processor.

During reset, the ACK line is internally pulled high with the equivalent of an internal 20K ohm resistor and is held high with an internal keeper latch. It is not necessary to use an external pull-up resistor on the ACK line during booting or at any other time.

The RBWS and RBAM fields of the WAIT register are initialized to perform asynchronous access and generate seven wait states (eight cycles total) for the EPROM access in external memory space. Note that wait states defined for boot memory are applied to $\overline{\text{BMS}}$ asserted accesses.

Table 5-3 shows how DMA channel 10 parameter registers are initialized at reset. The count register (CEP0) is initialized to 0x0100 to transfer 256 words to internal memory. The external count register (ECEP0), used when external addresses (BMS space) are generated by the DMA controller, is initialized to 0x0600 (0x0100 words at six bytes per word). The DMAC10 control register is initialized to 0x00 0561.

The default value sets up external port transfers as follows:

- DEN = 1, external port enabled
- MSWF = 0, LSB first
- PMODE = 101, 8-bit to 48-bit packing, Master = 1
- DTYPE = 1, three column data

Table 5-3. DMA Channel 10 Parameter Registers for EPROM Booting

Parameter Register	Initialization Value
IIEP0	0x40000
IMEP0	Uninitialized (increment by 1 is automatic)
CEP0	0x100 (256-instruction words)
CPEP0	Uninitialized
GPEP0	Uninitialized
EIEP0	0x800000
EMEP0	Uninitialized (increment by 1 is automatic)
ECEP0	0x600 (256 words x 6 bytes/word)

The following sequence occurs at system start-up, when the processor $\overline{\text{RESET}}$ input goes inactive.

1. The processor goes into an idle state, identical to that caused by the IDLE instruction. The program counter (PC) is set to address 0x40004.
2. The DMA parameter registers for channel 10 are initialized as shown in Table 5-3.
3. The $\overline{\text{BMS}}$ pin becomes the boot EPROM chip select.
4. 8-bit master mode DMA transfers from EPROM to the first internal memory address on the external port data bus lines 23–16.
5. The external address lines (ADDR23–0) start at 0x800000 and increment after each access.
6. The $\overline{\text{RD}}$ strobe asserts as in a normal memory access with seven wait states (eight cycles).

The processor's DMA controller reads the 8-bit EPROM words, packs them into 48-bit instruction words, and transfers them to internal memory until 256 words have been loaded. The EPROM is automatically selected by the $\overline{\text{BMS}}$ pin; other memory select pins are disabled.

The master DMA internal and external count registers (ECEP0/CEP0) decrement after each EPROM transfer. When both counters reach zero, the following wake-up sequence occurs:

1. DMA transfers stop.
2. External port DMA channel 10 interrupt (EP0I) is activated.
3. The $\overline{\text{BMS}}$ pin is deactivated, and normal external memory selects are activated.
4. The processor vectors to the EP0I interrupt vector at 0x40050.

At this point, the processor has completed its boot and is executing instructions normally. The first instruction at the EP0I interrupt vector location, address 0x40050, should be an RTI (return from interrupt). This process returns execution to the reset routine at location 0x40005 where normal program execution can resume. After reaching this point, a program can write a different service routine at the EP0I vector location 0x40050.

Host Boot Mode

The processor can boot from a host processor through the external port. Host booting is selected when the EBOOT and LBOOT inputs are low and $\overline{\text{BMS}}$ is high. Configured for host booting, the processor enters the slave mode after reset and waits for the host to download the boot program.

The DMAC10 control register is initialized for booting, packing boot data into 48-bit instructions. Channel 10 of the IO processor's DMA controller is used to transfer instructions to internal memory. Processors accept data from 8- or 16-bit host microprocessor (or other external devices).

After the boot process loads 256 words into memory locations 0x40000 through 0x400FF, the processor begins executing instructions. Because most processor programs require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the application. VisualDSP++ includes loading routines (boot kernels) that can load entire programs; refer to "ADSP-21161 Processor Boot Kernels" on page 5-16 for more information.

Refer to the *ADSP-21161 SHARC DSP Hardware Reference* for detailed information on DMA and system configurations.

DMA channel differences between the ADSP-21161 and previous SHARC family processors (ADSP-2106x) account for boot differences. Even with these differences, the ADSP-21161 processors support the same boot capability and configuration as the ADSP-2106x processors. The DMAC10 register default values differ because the ADSP-21161 processor has additional parameters and different DMA channel assignments. Host boot mode uses EPB0, DMA channel 10.

The processor determines the boot mode at reset from the EBOOT, LBOOT, and $\overline{\text{BMS}}$ pin inputs. When EBOOT=0, LBOOT=0, and BMS=1, the processor boots from a host through the external port. Refer to Table 5-1 and Table 5-2 on page 5-4 for boot mode selection.

When using any of the power-up boot modes, address 0x40004 should not contain a valid instruction. Because it is not executed during the boot sequence, place a NOP or IDLE instruction at this location.

During reset, the processor ACK line is internally pulled high with an equivalent 20K ohm resistor and is held high with an internal keeper latch. It is not necessary to use an external pull-up resistor on the ACK line during booting or at any other time.

Table 5-4 shows how the DMA channel 10 parameter registers are initialized at reset for host boot. The internal count register (CEP0) is initialized to 0x0100 to transfer 256 words to internal memory. The DMAC10 control register is initialized to 0000 0161.

The default value sets up external port transfers as follows:

- DEN = 1, external port enabled
- MSWF = 0, LSB first
- PMODE = 101, 8-bit to 48-bit packing
- DTYPE = 1, three column data

Table 5-4. DMA Channel 10 Parameter Register for Host Boot

Parameter Register	Initialization Value
IIEP0	0x0004 0000
IMEP0	Uninitialized (increment by 1 is automatic)
CEP0	0x0100 (256-instruction words)
CPEP0	Uninitialized
GPEP0	Uninitialized
EIEP0	Uninitialized
EMEP0	Uninitialized
ECEP0	Uninitialized

At system start-up, when the processor $\overline{\text{RESET}}$ input goes inactive, the following sequence occurs.

1. The processor goes into an idle state, identical to that caused by the IDLE instruction. The program counter (PC) is set to address 0x40004.
2. The DMA parameter registers for channel 10 are initialized as shown in Table 5-4.

3. The host uses HBR and CS to arbitrate for the bus.
4. The host can write to SYSCON (if HBG and READY are returned) to change boot width from default.
5. The host writes boot information to external port buffer 0.

The slave DMA internal count register (CEPO) decrements after each transfer. When CEPO reaches zero, the following wake-up sequence occurs:

1. The DMA transfers stop.
2. The external port DMA channel 10 interrupt (EPOI) is activated.
3. The processor vectors to the EPOI interrupt vector at 0x40050.

At this point, the processor has completed its boot mode and is executing instructions normally. The first instruction at the EPOI interrupt vector location, address 0x40050, should be an RTI (return from interrupt). This process returns execution to the reset routine at location 0x40005 where normal program execution can resume. After reaching this point, a program can write a different service routine at the EPOI vector location 0x40050.

Link Port Boot Mode

Link port boot uses DMA channel 8 of the IO processor to transfer instructions to internal memory. In this boot mode, the processor receives 4-bit wide data in link buffer 0.

After the boot process loads 256 words into memory locations 0x40000 through 0x400FF, the processor begins to execute instructions. Because most processor programs require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the application. VisualDSP++ includes loading routines (boot kernels)

that load an entire program through the selected port; refer to "ADSP-21161 Processor Boot Kernels" on page 5-16 for more information.

Refer to the *ADSP-21161 SHARC DSP Hardware Reference* for detailed information on DMA and system configurations.

DMA channel differences between the ADSP-21161 and previous SHARC family processors (ADSP-2106x) account for boot differences. Even with these differences, the ADSP-21161 processors support the same boot capabilities and configuration as the ADSP-2106x processors.

The processor determines the boot mode at reset from the `EBOOT`, `LBOOT` and $\overline{\texttt{BMS}}$ pin inputs. When `EBOOT=0`, `LBOOT=1`, and `BMS=1`, the processor boots through the link port. For information on boot mode selection, see Table 5-1 and Table 5-2 on page 5-4.

When using any of the power-up booting modes, address `0x40004` should not contain a valid instruction. Because it is not executed during the boot sequence, place a `NOP` or `IDLE` instruction at this location.

In link port boot, the processor gets boot data from another processor link port or 4-bit wide external device after system power-up.

The external device must provide a clock signal to the link port assigned to link buffer 0. The clock can be any frequency up to the processor clock frequency. The clock falling edges strobe the data into the link port. The most significant 4-bit nibble of the 48-bit instruction must be downloaded first.

Table 5-5 shows how the DMA channel 8 parameter registers are initialized at reset. The count register (`CLB0`) is initialized to `0x0100` to transfer 256 words to internal memory. The `LCTL` register is overridden during link port boot to allow link buffer 0 to receive 48-bit data.

Table 5-5. DMA Channel 8 Parameter Register for Link Port Boot

Parameter Register	Initialization Value
IILB0	0x0004 0000
IMLB0	Uninitialized (increment by 1 is automatic)
CLB0	0x0100 (256-instruction words)
CPLB0	Uninitialized
GPLB0	Uninitialized

In systems where multiple processors are not connected by the parallel external bus, booting can be accomplished from a single source through the link ports. To simultaneously boot all the processors, make a parallel common connection to link buffer 0 on each of the processors. If a daisy chain connection exists between the processors' link ports, each processor can boot the next processor in turn. Link buffer 0 must always be used for booting.

SPI Port Boot Mode

Serial peripheral interface (SPI) port booting uses DMA channel 8 of the IO processor to transfer instructions to internal memory. In this boot mode, the processor receives 8-bit wide data in the SPIRx register.

During the boot process, the program loads 256 words into memory locations 0x40000 through 0x400FF. The processor subsequently begins executing instructions. Because most processor programs require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the application. VisualDSP++ includes loading routines (boot kernels) which load an entire program through the selected port. See "ADSP-21161 Processor Boot Kernels" on page 5-16 for more information.

Refer to the *ADSP-21161 SHARC DSP Hardware Reference* for detailed information on DMA and system configurations. For information about SPI slave booting, refer to *EE-177: SHARC SPI Booting*, located on the Analog Devices processor Web site.

The processor determines the boot mode at reset from the EBOOT, LBOOT, and $\overline{\text{BMS}}$ pin inputs. When EBOOT=0, LBOOT=1, and BMS=0, the processor boots through its SPI port. For information on the boot mode selection, see Table 5-1 and Table 5-2 on page 5-4.

When using any of the power-up booting modes, address 0x40004 should not contain a valid instruction. Because it is not executed during the boot sequence, place a NOP or IDLE instruction placed at this location.

For SPI port boot, the processor gets boot data after system power-up from another processor's SPI port or another SPI compatible device.

Table 5-6 shows how the DMA channel 8 parameter registers are initialized at reset. The SPI control register (SPICTL) is configured to 0x0A001F81 upon reset during SPI boot.

This configuration sets up the SPIRx register for 32-bit serial transfers. The SPIRx DMA channel 8 parameter registers are configured to DMA in 0x180 32-bit words into internal memory normal word address space starting at 0x40000. Once the 32-bit DMA transfer completes, the data is accessed as 3 column, 48-bit instructions. The processor executes a 256 word (0x100) boot kernel upon completion of the 32-bit, 0x180 word DMA.

For 16-bit SPI hosts, two words are shifted into the 32-bit receive shift register before a DMA transfer to internal memory occurs. For 8-bit SPI hosts, four words are shifted into the 32-bit receive shift register before a DMA transfer to internal memory occurs.

Table 5-6. DMA Channel 8 Parameter Register for SPI Port Boot

Parameter Register	Initialization Value
IISRX	0x0004 0000
IMSRX	Uninitialized (increment by 1 is automatic)
CSRX	0x0180 (256-instruction words)
GPSRX	Uninitialized

No-Boot Mode

No-boot mode causes the processor to start fetching and executing instructions at address 0x200004 in external memory space. In no-boot mode, the processor does not boot-load and all DMA control and parameter registers are set to their default initialization values.The loader utility does not produce the code for no-boot execution.

ADSP-21161 Processor Boot Kernels

The boot-loading process starts with a transfer of the boot kernel program into the processor memory. The boot kernel sets up the processor and loads boot data. After the boot kernel finishes initializing the rest of the system, the boot kernel loads boot data over itself with a final DMA transfer.

Four boot kernels ship with VisualDSP++; refer to Table 5-7.

Table 5-7. ADSP-21161 Default Boot Kernel Files

PROM Booting	Link Booting	Host Booting	SPI Booting
161_prom.dxe	161_link.dxe	161_host.dxe	161_spi.dxe

Boot kernels are loaded at processor reset into the seg_ldr memory segment, which is defined in the 161_ldr.ldf. The file is stored in the <*install_path*>\211xx\ldr directory of VisualDSP++.

ADSP-21161 Processor Boot Streams

The loader utility produces the boot stream in blocks and inserts header words at the beginning of data blocks in the loader (`.ldr`) file. The boot kernel uses header words to properly place data and instruction blocks into processor memory. The header format for PROM, host, and link boot-loader files is as follows.

```
0x000000000DDDD
0xAAAAAAAALLLL
```

In the above example, `D` is a data block type tag, `A` is a block start address, and `L` is a block word length.

For single-processor systems, the data block header has three 32-bit words in SPI boot mode, as follows.

`0xLLLLLLLL`	First word. Data word length or data word count of the data block.
`0xAAAAAAAA`	Second word. Data block start address.
`0x000000DD`	Third word. Tag of data block type.

The boot kernel examines the tag to determine the type of data or instruction being loaded. Table 5-8 lists the ADSP-21161N processor block tags.

Table 5-8. ADSP-21161N Processor Block Tags

Tag Number	Block Type	Tag Number	Block Type
0x0000	final init	0x000E	init pm48
0x0001	zero dm16	0x000F	zero dm64
0x0002	zero dm32	0x0010	init dm64
0x0003	zero dm40	0x0012	init pm64
0x0004	init dm16	0x0013	init pm8 ext
0x0005	init dm32	0x0014	init pm16 ext

Table 5-8. ADSP-21161N Processor Block Tags (Cont'd)

Tag Number	Block Type	Tag Number	Block Type
0x0007	zero pm16	0x0015	init pm32 ext
0x0008	zero pm32	0x0016	init pm48 ext
0x0009	zero pm40	0x0017	zero pm8 ext
0x000A	zero pm48	0x0018	zero pm16 ext
0x000B	init pm16	0x0019	zero pm32 ext
0x000C	init pm32	0x001A	zero pm48 ext
0x0011	zero pm64		

Boot Kernel Modification and Loader Issues

Some systems require boot kernel customization. In addition, the operation of other tools (such as the C/C++ compiler) is influenced by whether the loader utility is used.

If you do not specify a boot kernel file via the **Load** page of the **Project Options** dialog box in VisualDSP++ (or via the -l kernelfile command-line switch), the loader utility places a default boot kernel in the loader output file (see "ADSP-21161 Processor Boot Kernels" on page 5-16) based on the specified boot mode.

Rebuilding a Boot Kernel File

If you modify the boot kernel source (`.asm`) file by inserting correct values for your system, you must rebuild the boot kernel (`.dxe`) before generating the boot-loadable (`.ldr`) file. The boot kernel source file contains default values for the `SYSCON` register. The `WAIT`, `SDCTL`, and `SDRDIV` initialization code is in the boot kernel file comments.

To Modify a Boot Kernel Source File

1. Copy the applicable boot kernel source file (`161_link.asm`, `161_host.asm`, `161_prom.asm`, or `161_spi.asm`).
2. Apply the appropriate initializations of the `SYSCON` and `WAIT` registers.

After modifying the boot kernel source file, rebuild the boot kernel (`.dxe`) file. Do this from the VisualDSP++ IDDE (refer to VisualDSP++ online Help for details), or rebuild the boot kernel file from the command line.

Rebuilding a Boot Kernel Using Command Lines

Rebuild a boot kernel using command lines as follows.

EPROM Boot. The default boot kernel source file for EPROM booting is `161_prom.asm`. After copying the default file to `my_prom.asm` and modifying it to suit your system, use the following command lines to rebuild the boot kernel.

```
easm21k -proc ADSP-21161 my_prom.asm
linker -T 161_ldr.ldf my_prom.doj
```

Host Boot. The default boot kernel source file for host booting is `161_host.asm`. After copying the default file to `my_host.asm` and modifying it to suit your system, use the following command lines to rebuild the boot kernel.

```
easm21k -proc ADSP-21161 my_host.asm
linker -T 161_ldr.ldf my_host.doj
```

Link Boot. The default boot kernel source file for link booting is `161_link.asm`. After copying the default file to `my_link.asm` and modifying it to suit your system, use the following command lines to rebuild the boot kernel.

```
easm21k -proc ADSP-21161 my_link.asm
linker -T 161_ldr.ldf my_link.doj
```

SPI Boot. The default boot kernel source file for link booting is `161_SPI.asm`. After copying the default file to `my_SPI.asm` and modifying it to suit your system, use the following command lines to rebuild the boot kernel:

```
easm21k -proc ADSP-21161 my_SPI.asm
linker -T 161_ldr.ldf my_SPI.doj
```

Loader File Issues

If you modify the boot kernel for the EPROM, host, SPI, or link booting modes, ensure that the `seg_ldr` memory segment is defined in the `.ldf` file. Refer to the source of this memory segment in the `.ldf` file located in the `…\ldr\` directory of the of the target processor.

Because the loader utility uses the address of `0x40004` for the first location of the reset vector during the boot-load process, avoid placing code at this address. When using any of the processor's power-up boot modes, ensure that this address does not contain a critical instruction. Because this address is not executed during the booting sequence, place a `NOP` or `IDLE` in this location. The loader utility generates a warning if the vector address `0x40004` does not contain `NOP` or `IDLE`.

When using VisualDSP++ to create the loader file, specify the name of the customized boot kernel executable in the **Kernel file** box on the **Load** page of the **Project Options** dialog box.

ADSP-21161 Processor Interrupt Vector Table

If the ADSP-21161 processor is booted from an external source (EPROM, host, link port, or SPI), the interrupt vector table is located in internal memory. If the processor is not booted and executes from external memory (no-boot mode), the vector table must be located in external memory.

The `IIVT` bit in the `SYSCON` control register can be used to override the booting mode in determining where the interrupt vector table is located. If the processor is not booted (no-boot mode), setting `IIVT` to `1` selects an internal vector table, and setting `IIVT` to zero selects an external vector table. If the processor is booted from an external source (any boot mode other than no-boot), `IIVT` has no effect. The default initialization value of `IIVT` is zero.

ADSP-21161 Multi-Application (Multi-DXE) Management

Currently, the loader utility generates single-processor loader files for host, link, and SPI port boot. The loader utility supports multiprocessor EPROM boot only. The application code must be modified to properly set up multiprocessor booting in host, link, and SPI port boot modes.

There are two methods by which a multiprocessor system can be booted:

- Boot From a Single EPROM
- Sequential EPROM Boot

Regardless of the method, the processors perform the following steps.

1. Arbitrate for the bus
2. Upon becoming bus master, DMA the 256-word boot stream

3. Release the bus
4. Execute the loaded instructions

Boot From a Single EPROM

The loader utility can produce boot-loadable files that permit SHARC processors in a multiprocessor system to boot from a single EPROM. The $\overline{\texttt{BMS}}$ signals from each processor may be wire ORed together to drive the EPROM's chip select pin. Each processor can boot in turn, according to its priority. When the last processor has finished booting, it must inform the other processors (which may be in the idle state) that program execution can begin (if all processors are to begin executing instructions simultaneously).

When multiple processors boot from a single EPROM, the processors can boot identical code or different code from the EPROM. If the processors load differing code, use a jump table in the loader file (based on processor ID) to select the code for each processor.

Sequential EPROM Boot

Set the EBOOT pin of the processor with ID# of 1 high for EPROM booting. The other processors should be configured for host boot (EBOOT=0, LBOOT=0, and BMS=1), leaving them in the idle state at startup and allowing the processor with ID=1 to become bus master and boot itself. Connect the $\overline{\texttt{BMS}}$ pin of processor #1 only to the EPROM's chip select pin. When processor #1 has finished booting, it can boot the remaining processors by writing to their external port DMA buffer 0 (EPB0) via the multiprocessor memory space.

Processor ID Numbers

A single-processor system requires only one input (.dxe) file without any prefix and suffix to the input file name, for example:

```
elfloader -proc ADSP-21161 -bprom Input.dxe
```

A multiprocessor system requires a distinct processor ID number for each input file on the command line. A processor ID is provided via the -id#exe=*filename.dxe* switch, where # is 1 to 6.

In the following example, the loader utility processes the input file Input1.dxe for the processor with an ID of 1 and the input file Input2.dxe for the processor with an ID of 2.

```
elfloader -proc ADSP-21161 -bprom -id1exe=Input1.dxe
          -id2exe=Input2.dxe
```

If the executable for the # processor is identical to the executable of the *N* processor, the output loader file contains only one copy of the code from the input file, as directed by the command-line switch -id#ref=N used in the example:

```
elfloader -proc ADSP-21161 -bprom -id1exe=Input.dxe -id2ref=1
```

where 2 is the processor ID, and 1 is another processor ID referenced by processor 2.

The loader utility points the id(2)exe loader jump table entry to the id(1)exe image, effectively reducing the size of the loader file.

ADSP-21161 Processor Loader Guide

Loader operations depend on the loader options, which control how the loader utility processes executable files. You select features such as boot modes, boot kernels, and output file formats via the options. The options are specified on the loader utility's command line or via the **Load** page of the **Project Options** dialog box in the VisualDSP++ environment.

The **Load** page consists of multiple panes. For information specific to the ADSP-21161 processor, refer to the VisualDSP++ online help for that processor. When you open the **Load** page, the default loader settings for the selected processor are already set. Use the **Additional Options** box to enter options that have no dialog box equivalent.

Option settings on the **Load** page correspond to switches displayed on the command line.

These sections describe how to produce a bootable loader (`.ldr`) file:

- "Using ADSP-21161 Loader Command Line" on page 5-25
- "Using VisualDSP++ Interface (Load Page)" on page 5-32

Using ADSP-21161 Loader Command Line

Use the following syntax for the ADSP-21161 loader command line.

```
elfloader inputfile -proc ADSP-21161 -switch [-switch…]
```

where:

- *inputfile*—Name of the executable file (`.dxe`) to be processed into a single boot-loadable file. An input file name can include the drive and directory. Enclose long file names within straight quotes, "`long file name`".
- `-proc ADSP-21161`—Part number of the processor for which the loadable file is built. The `-proc` switch is mandatory.
- *-switch …*—One or more optional switches to process. Switches select operations and boot modes for the loader utility. A list of all switches and their descriptions appear in Table 5-10 on page 5-28.

Command-line switches are not case-sensitive and placed on the command line in any order.

Single-Processor Systems

The following command line,

```
elfloader Input.dxe -bSPI -proc ADSP-21161
```

runs the loader utility with:

- `Input.dxe`—Identifies the executable file to process into a boot-loadable file for a single-processor system. Note that the absence of the `-o` switch causes the output file name to default to `Input.ldr`.

- `-bSPI`—Specifies SPI port booting as the boot type for the boot-loadable file.
- `-proc ADSP-21161`—Specifies ADSP-21161 as the target processor.

Multiprocessor Systems

The following command line,

```
elfloader -proc ADSP-21161 -bprom -id1exe=Input1.dxe
          -id2exe=Input2.dxe
```

runs the loader utility with:

- `-proc ADSP-21161`—Specifies ADSP-21161 as the target processor.
- `-bprom`—Specifies EPROM booting as the boot type for the boot-loadable file.
- `-id1exe=Input1.dxe`—Identifies `Input1.dxe` as the executable file to process into a boot-loadable file for a processor with ID of `1` (see "Processor ID Numbers" on page 5-23).
- `-id2exe=Input2.dxe`—Identifies `Input2.dxe`. as the executable file to process into a boot-loadable file for a processor with ID of `2` (see "Processor ID Numbers" on page 5-23).

File Searches

File searches are important in loader processing. The loader utility supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur as described on page 1-17.

File Extensions

Some loader switches take a file name as an optional parameter. Table 5-9 lists the expected file types, names, and extensions.

Table 5-9. File Extensions

Extension	File Description
`.dxe`	Executable files and boot kernel files. The loader utility recognizes overlay memory files (`.ovl`) and shared memory files (`.sm`) but does not expect these files on the command line. Place `.ovl` and `.sm` files in the same directory as the `.dxe` file that refers to them so the loader utility can find them when processing the `.ldr` file. The `.ovl` and `.sm` files can also be placed in the `.ovl` and `.sm` file output directory specified in the `.ldf` file.
`.ldr`	Loader output file

Loader Command-Line Switches

Table 5-10 is a summary of the ADSP-21161 loader switches.

Table 5-10. ADSP-21161 Loader Command Line Switches

Switch	Description

`-bprom`
`-bhost`
`-blink`
`-bspi`
	Specifies the boot mode. The `-b` switch directs the loader utility to prepare a boot-loadable file for the specified boot mode. The valid modes (boot types) are PROM, host, link, and SPI.
	If the switch does not appear on the command line, the default is `-bprom`.
	To use a custom boot kernel, the boot mode selected with the `-b` switch must correspond with the boot kernel selected with the -l kernelfile switch. Otherwise, the loader utility automatically selects a default boot kernel based on the selected boot type (see "ADSP-21161 Processor Boot Kernels" on page 5-16).

`-e`*`filename`*	Except shared memory. The `-e` switch omits the specified shared memory (`.sm`) file from the output loader file. Use this option to omit the shared parts of the executable file intended to boot a multiprocessor system. To omit multiple `.sm` files, repeat the switch and its parameter multiple times on the command line. For example, to omit two files, use: `-efileA.SM -efileB.SM`. In most cases, it is not necessary to use the `-e` switch: the loader utility processes the `.sm` files efficiently (includes a single copy of the code and data from each `.sm` file in a loader file).

`-fhex`
`-fASCII`
`-fbinary`
`-finclude`
`-fS1`
`-fS2`
`-fS3`
	Specifies the format of the boot-loadable file (Intel hex-32, ASCII, include, binary, S1, S2, and S3 (Motorola S-records). If the `-f` switch does not appear on the command line, the default boot file format is hex for PROM, and ASCII for host, link, or `SPI`.
	Available formats depend on the boot mode selection (`-b` switch):
	• For a PROM boot, select a hex-32, S1, S2, S3, ASCII, or include format.
	• For host or link boot, select an ASCII, binary, or include format.
	• For SPI boot, select an ASCII or binary format.

Table 5-10. ADSP-21161 Loader Command Line Switches (Cont'd)

Switch	Description

`-h`
or
`-help`
	Command-line help. Outputs the list of command-line switches to standard output and exits.
	Combining the `-h` switch with `-proc ADSP-21161`; for example, `elfloader -proc ADSP-21161 -h`, yields the loader syntax and switches for the ADSP-21161 processors. By default, the `-h` switch alone provides help for the loader driver.

`-hostwidth #`	Sets up the word width for the `.ldr` file. By default, the word width for PROM and host is 8, for link is 16, and for SPI is 32. The valid word widths for the various boot modes are: • PROM—8 for hex or ASCII format, 8 or 16 for include format • host—8 or 16 for ASCII or binary format, 16 for include format • link—16 for ASCII, binary, or include format • SPI—8, 16, or 32 for Intel hex 32 or ASCII format
`-id#exe=`*`filename`*	Specifies the processor ID. The `-id#exe=` switch directs the loader utility to use the processor ID (`#`) for the corresponding executable file (*`filename`*) when producing a boot-loadable file for EPROM boot of a multiprocessor system. This switch is used only to produce a boot-loadable file that boots multiple processors from a single EPROM. Valid values for `#` are `1`, `2`, `3`, `4`, `5`, and `6`. Do not use this switch for single-processor systems. For single-processor systems, use *`filename`* as a parameter without a switch. For more information, refer to "Processor ID Numbers" on page 5-23.
`-id#ref=`*`N`*	Points the processor ID (`#`) loader jump table entry to the ID (*`N`*) image. If the executable file for the (`#`) processor is identical to the executable of the (*`N`*) processor, the switch can be used to set the PROM start address of the processor with ID of # to be the same as for the processor with ID of *`N`*. This effectively reduces the size of the loader file by providing a single copy of an executable to two or more processors in a multiprocessor system. For more information, refer to "Processor ID Numbers" on page 5-23.

Table 5-10. ADSP-21161 Loader Command Line Switches (Cont'd)

Switch	Description
`-l kernelfile`	Directs the loader utility to use the specified *kernelfile* as the boot-loading routine in the output boot-loadable file. The boot kernel selected with this switch must correspond to the boot mode selected with the `-b` switch. If the `-l` switch does not appear on the command line, the loader utility searches for a default boot kernel file. Based on the boot mode (`-b` switch), the loader utility searches in the processor-specific loader directory for the boot kernel file as described in "ADSP-21161 Processor Boot Kernels" on page 5-16.
`-o filename`	Directs the loader utility to use the specified *filename* as the name for the loader output file. If not specified, the default name is *inputfile*`.ldr`.
`-noZeroBlock`	The `-noZeroBlock` switch directs the loader utility not to build zero blocks.
`-paddress`	Directs the loader utility to start the boot-loadable file at the specified address in the EPROM. This EPROM address corresponds to `0x8000000` on the ADSP-21161 processor. If the `-p` switch does not appear on the command line, the loader utility starts the EPROM file at address `0x0`.
`-proc ADSP-21161`	Specifies the processor. This is a mandatory switch.

Table 5-10. ADSP-21161 Loader Command Line Switches (Cont'd)

Switch	Description
`-si-revision #|none|any`	The `-si-revision {#|none|any}` switch provides a silicon revision of the specified processor. The switch parameter represents a silicon revision of the processor specified by the `-proc` *`processor`* switch. The parameter takes one of three forms: • The `none` value indicates that the VisualDSP++ ignores silicon errata. • The # value indicates one or more decimal digits, followed by a point, followed by one or two decimal digits. Examples of revisions are: `0.0`; `1.12`; `23.1`. Revision `0.1` is distinct from and "lower" than revision `0.10`. The digits to the left of the point specify the chip tape-out number; the digits to the right of the point identify the metal mask revision number. The number to the right of the point cannot exceed decimal `255`. • The `any` value indicates that VisualDSP++ produces an output file that can be run at any silicon revision. The switch generates either a warning about any potential anomalous conditions or an error if any anomalous conditions occur. In the absence of the silicon revision switch, the loader utility selects the greatest silicon revision it is aware of, if any. In the absence of the switch parameter (a valid revision value)—`-si-revision` alone or with an invalid value—the loader utility generates an error.
`-t#`	(Host boot type only) Specifies timeout cycles. The `-t` switch (for example, `-t100`) limits the number of cycles that the processor spends initializing external memory with zeros. Valid values range from `3` to `32765` cycles; `32765` is the default value. The timeout value (#) is related directly to the number of cycles the processor locks the bus for boot-loading, instructing the processor to lock the bus for no more than two times the timeout number of cycles. When working with a fast host that cannot tolerate being locked out of the bus, use a relatively small timeout value.

Table 5-10. ADSP-21161 Loader Command Line Switches (Cont'd)

Switch	Description
`-v`	Outputs verbose loader messages and status information as the loader utility processes files.
`-version`	Directs the loader utility to show its version information. Type `elfloader -version` to display the version of the loader drive. Add the `-proc` switch, for example, `elfloader -proc ADSP-21161 -version` to display version information of both loader drive and SHARC loader.

Using VisualDSP++ Interface (Load Page)

After selecting a **Loader file** as the target type on the **Project** page in VisualDSP++ **Project Options** dialog box, modify the default options on the **Load: Processor** page (also called loader property page). Click **OK** to save the selections. Selecting **Build Project** from the **Project** menu generates a loader file. For information relative to a specific processor, refer to the VisualDSP++ online help for that processor.

VisualDSP++ invokes the `elfloader` utility to build the output file. The **Load** page buttons and fields correspond to loader command-line switches and parameters (see Table 5-10 on page 5-28). Use the **Additional Options** box to enter options that do not have dialog box equivalents.

6 LOADER FOR ADSP-2126X/2136X/2137X/2146X SHARC PROCESSORS

This chapter explains how the loader utility (`elfloader.exe`) is used to convert executable (`.dxe`) files into boot-loadable files for the ADSP-2126x, ADSP- 2136x, ADSP-2137x, and ADSP-2146x SHARC processors.

Refer to "Introduction" on page 1-1 for the loader utility overview; the introductory material applies to all processor families. Refer to "Loader for ADSP-2106x/21160 SHARC Processors" on page 4-1 for information about the ADSP-21060, ADSP-21061, ADSP-21062, ADSP-21065L, and ADSP-21160 processors. Refer to "Loader for ADSP-21161 SHARC Processors" on page 5-1 for information about the ADSP-21161 processors.

Loader operations specific to the ADSP-2126x/2136x/2137x/2146x SHARC processors are detailed in the following sections.

- "ADSP-2126x/2136x/2137x/2146x Processor Booting" Provides general information about various booting modes, including information about boot kernels.
- "ADSP-2126x/2136x/2137x/2146x Processor Loader Guide" Provides reference information about the graphical user interface, command-line syntax, and switches.

ADSP-2126x/2136x/2137x/2146x Processor Booting

The ADSP-2126x, ADSP-2136x, ADSP-2137x, and ADSP-2146x processors can be booted from an external PROM memory device via the parallel port (PROM mode) or via the serial peripheral interface (SPI slave, SPI flash, or SPI master mode). In no-boot mode, the processor is booted from the internal ROM (only available on some processors).

- In parallel port boot mode, the loader output file (`.ldr`) is stored in an 8-bit wide parallel PROM device and fetched by the processor.

On the ADSP-2126x/2136x/2137x/2146x processors, whether supporting multiprocessing or not, there is no ID lookup table between the kernel and the rest of the application.

- In SPI slave boot mode, the loader file is transmitted to the processor by a host processor configured as an SPI master.
- There are three cases for the SPI master boot mode: SPI master (no address), SPI PROM (16-bit address), and SPI flash (24-bit address). The difference between the these modes is the way the slave device sends the first word of the `.ldr` file. In SPI PROM and SPI flash boot modes, the `.ldr` file is stored in a passive memory device and fetched by the processor. In SPI master, the `.ldr` file is transmitted to the processor by a host processor configured as an SPI slave.
- In no-boot mode, the processor fetches and executes instructions directly from the external memory, bypassing the boot kernel entirely. The loader utility does not produce a file supporting the no-boot mode.

Software developers who use the loader utility should be familiar with the following operations.

- "Power-Up Booting Process" on page 6-3
- "Boot Mode Selection" on page 6-4
- "ADSP-2126x/2136x/2137x/2146x Processors Boot Modes" on page 6-5
- "ADSP-2126x/2136x/2137x/2146x Processors Boot Kernels" on page 6-19
- "ADSP-2126x/2136x/2137x/2146x Processors Interrupt Vector Table" on page 6-22
- "ADSP-2126x/2136x/2137x/2146x Processor Boot Streams" on page 6-23

Power-Up Booting Process

The ADSP-2126x, ADSP-2136x, ADSP-2137x, and ADSP-2146x processors include a hardware feature that boot-loads a small, 256-instruction, program into the processor's internal memory after power-up or after the chip reset. These instructions come from a program called a boot kernel. When executed, the boot kernel facilitates booting of user application code. The combination of the boot kernel and application code comprise the boot-loadable (`.ldr`) file.

At power-up, after the chip reset, the booting process includes the following steps.

1. Based on the boot type, an appropriate DMA channel is automatically configured for a 384-word (32-bit) transfer or a 256-word (48-bit) transfer. This transfer boot-loads the boot kernel program into the processor memory.

2. The boot kernel runs and loads the application executable code and data.
3. The boot kernel overwrites itself with the first 256 (48-bit) words of the application at the end of the booting process. After that, the application executable code starts running.

The boot type selection directs the system to prepare the appropriate boot kernel.

Boot Mode Selection

Unlike earlier SHARC processors, ADSP-2126x/2136x/2137x/2146x processors do not have a boot memory select ($\overline{\text{BMS}}$) pin. On the ADSP-2126x/2136x/2137x/2146x processor, the boot type is determined by sampling the state of the BOOT_CFGx pins, as described in Table 6-1 and Table 6-2. A description of each boot type follows in "ADSP-2126x/2136x/2137x/2146x Processors Boot Modes".

Table 6-1. ADSP-2126x/2136x/2137x Boot Mode Pins

BOOT_CFG[1–0]	Boot Mode	Boot Mode Selection
00	SPI slave	-bspislave
01	SPI master (SPI flash, SPI PROM, or a host processor via SPI master mode)	-bspiflash -bspiprom -bspimaster
10	EPROM boot via the parallel port	-bprom
11	Internal boot (not available on all ADSP-2126x processors)	Does not use the loader utility

Table 6-2. ADSP-2146x Boot Mode Pins

BOOT_CFG[2–0]	Boot Mode	Boot Mode Selection
000	SPI slave	-bspislave
001	SPI master (SPI flash, SPI PROM, or a host processor via SPI master mode)	-bspiflash -bspiprom -bspimaster
010	AMI user boot (for 8-bit flash memory boot)	-bprom
011	Reserved	N/A
100	Link port 0 boot	-blink
101	Reserved	N/A

ADSP-2126x/2136x/2137x/2146x Processors Boot Modes

The following sections describe the ADSP-2126x/2136x/2137x/2146x processor boot types:

- "PROM Boot Mode" on page 6-5
- "SPI Port Boot Modes" on page 6-7
- "Internal Boot Mode" on page 6-17

PROM Boot Mode

The ADSP-2126x/2136x/2137x/2146x processors support an 8-bit boot mode through the parallel port. This mode is used to boot from external 8-bit-wide memory devices. The processor is configured for 8-bit boot mode when the BOOT_CFG1-0 pins = 10 orBOOT_CFG2-0 pins = 010. When configured for parallel booting, the parallel port transfers occur with the default bit settings for the PPCTL register (shown in Table 6-3).

Table 6-3. PPCTL Register Settings for PROM Boot Mode

Bit	Setting
PPALEPL	= 0; ALE is active high
PPEN	= 1
PPDUR	= 10111; (23 core clock cycles per data transfer cycle)
PPBHC	= 1; insert a bus hold cycle on every access
PP16	= 0; external data width = 8 bits
PPDEN	= 1; use DMA
PPTRAN	= 0; receive (read) DMA
PPBHD	= 0; buffer hang enabled

The parallel port DMA channel is used when downloading the boot kernel information to the processor. At reset, the DMA parameter registers are initialized to the values listed in Table 6-4.

Table 6-4. Parameter Register Settings for PROM Boot Mode

Parameter Register	Initialization Value	Comment
PPCTL	0x0000 016F	See Table 6-3
IIPP	0 for the ADSP-2126x processors; 0x10000 for the ADSP-2136x/2146x processors	The offset from internal memory normal word starting address of 0x80000
ICPP	0x180 (384)	The number of 32-bit words that are equivalent to 256 instructions
IMPP	0x01	
EIPP	0x00	
ECPP	0x600	The number of bytes in 0x100 48-bit instructions
EMPP	0x01	

Packing Options for External Memory

For the ADSP-2126x processors, the external memory address ranges are `0x1000000–0x2FFFFFF`. For the ADSP-21362/21363/21364/21365/21366 processors, the external memory address ranges are `0x1200000–0x1203FFF`. For the ADSP-21367/21368/21369/2137x and ADSP-2146x processors, the external PM memory address ranges are `0x200000–0x23FFFF`. The parallel port automatically packs internal 32-bit words to either 8-bit or 16-bit words for external memory. These are the only widths supported. The `WIDTH()` command in the linker specifies which packing mode should be used to initialize the external memory: `WIDTH(8)` for 8-bit memory or `WIDTH(16)` for 16-bit memory.

The loader utility packs the external memory data from the `.dxe` file according to the linker's `WIDTH()` command. The loader utility unpacks the data from the executable file and packs the data again in the loader file if the data is packed in the `.dxe` file due to the packing command in the linker description (`.ldf`) file.

Packing and Padding Details

For `ZERO_INIT` sections in a `.dxe` file, no data packing or padding in the `.ldr` file is required because only the header itself is included in the `.ldr` file. However, for other section types, additional data manipulation is required. It is important to note that in *all* cases, the word count placed into the block header in the loader file is the original number of words. That is, the word count does *not* include the padded word.

SPI Port Boot Modes

The ADSP-2126x/2136x/2137x/2146x SHARC processor supports booting from a host processor via serial peripheral interface slave mode (`BOOT_CFG1-0 = 00` or `BOOT_CFG2-0 = 000`), and booting from an SPI flash, SPI PROM, or a host processor via SPI master mode (`BOOT_CFG1-0 = 01`). SPI slave boot mode is discussed on page 6-8, and SPI master boot modes are discussed on page 6-10.

Both SPI boot modes support booting from 8-, 16-, or 32-bit SPI devices. In all SPI boot modes, the data word size in the shift register is hardwired to 32 bits. Therefore, for 8- or 16-bit devices, data words are packed into the shift register (RXSPI) to generate 32-bit words least significant bit (LSB) first, which are then shifted into internal memory.

For 16-bit SPI devices, two words shift into the 32-bit receive shift register (RXSR) before a DMA transfer to internal memory occurs. For 8-bit SPI devices, four words shift into the 32-bit receive shift register before a DMA transfer to internal memory occurs.

When booting, the ADSP-2126x/2136x/2137x/2146x processor expects to receive words into the RXSPI register seamlessly. This means that bits are received continuously without breaks in the CS link. For different SPI host sizes, the processor expects to receive instructions and data packed in a least significant word (LSW) format.

See the manual for the target SHARC processor peripherals for information on how data is packed into internal memory during SPI booting for SPI devices with widths of 32, 16, or 8 bits.

SPI Slave Boot Mode

In SPI slave boot mode, the host processor initiates the booting operation by activating the SPICLK signal and asserting the $\overline{\text{SPIDS}}$ signal to the active low state. The 256-word boot kernel is loaded 32 bits at a time, via the SPI receive shift register. To receive 256 instructions (48-bit words) properly, the SPI DMA initially loads a DMA count of 384 32-bit words, which is equivalent to 256 48-bit words.

The processor's $\overline{\text{SPIDS}}$ pin should not be tied low. When in SPI slave mode, including booting, the $\overline{\text{SPIDS}}$ signal is required to transition from high to low. SPI slave booting uses the default bit settings shown in Table 6-5.

Table 6-5. SPI Slave Boot Bit Settings

Bit	Setting	Comment
SPIEN	Set (= 1)	SPI enabled
MS	Cleared (= 0)	Slave device
MSBF	Cleared (= 0)	LSB first
WL	10, 32-bit SPI	Receive Shift register word length
DMISO	Set (= 1) MISO	MISO disabled
SENDZ	Cleared (= 0)	Send last word
SPIRCV	Set (= 1)	Receive DMA enabled
CLKPL	Set (= 1)	Active low SPI clock
CPHASE	Set (= 1)	Toggle SPICLK at the beginning of the first bit

The SPI DMA channel is used when downloading the boot kernel information to the processor. At reset, the DMA parameter registers are initialized to the values listed in Table 6-6.

Table 6-6. Parameter Register Settings for SPI Slave Boot

Parameter Register	Initialization Value	Comment
SPICTL	0x0000 4D22	
SPIDMAC	0x0000 0007	Enabled, RX, initialized on completion
IISPI	0x0008 0000	Start of block 0 normal word memory
IMSPI	0x0000 0001	32-bit data transfers
CSPI	0x0000 0180	

SPI Master Boot Modes

In SPI master boot mode, the ADSP-2126x/2136x/2137x/2146x processor initiates the booting operation by:

1. Activating the SPICLK signal and asserting the FLAG0 signal (on ADSP-2126x and ADSP-21362/21363/21364/21365/21366) or the SPI_FLAG0_O signal (routed by default to the DPI_PB05 pin on the ADSP-21367/21368/21369/2137x and ADSP-2146x) to the active low state to enable slave select
2. Writing the read command 0x03 and address 0x00 to the slave device

SPI master boot mode is used when the processor is booting from an SPI compatible serial PROM, serial flash, or slave host processor. The specifics of booting from these devices are discussed individually:

- "Booting From an SPI Flash" on page 6-16
- "Booting From an SPI PROM (16-bit address)" on page 6-16
- "Booting From an SPI Host Processor" on page 6-17

On reset, the interface starts up in SPI master mode performing a three hundred eighty-four 32-bit word DMA transfer.

SPI master booting uses the default bit settings shown in Table 6-7.

Table 6-7. SPI Master Boot Mode Bit Settings

Bit	Setting	Comment
SPIEN	Set (= 1)	SPI enabled
MS	Set (= 1)	Master device
MSBF	Cleared (= 0)	LSB first
WL	10	32-bit SPI receive shift register word length

Table 6-7. SPI Master Boot Mode Bit Settings (Cont'd)

Bit	Setting	Comment
DMISO	Cleared (= 0)	MISO enabled
SENDZ	Set (= 1)	Send zeros
SPIRCV	Set (= 1)	Receive DMA enabled
CLKPL	Set (= 1)	Active low SPI clock
CPHASE	Set (= 1)	Toggle SPICLK at the beginning of the first bit

The SPI DMA channel is used when downloading the boot kernel information to the processor. At reset, the DMA parameter registers are initialized to the values listed in Table 6-8.

Table 6-8. Parameter Registers Settings for SPI Master Boot

Parameter Register	Initialization Value	Comment
SPICTL	0x0000 5D06	
SPIBAUD	0x0064	CCLK/400 =500 KHz@ 200 MHz
SPIFLG	0xfe01	• FLAG0 on the ADSP-2126x and ADSP-21362/21363/21364/21365/2136 processors • SPI_FLAG0_0 is used as slave-select on the ADSP-21367/21368/21369/2137x and ADSP-2146x processors
SPIDMAC	0x0000 0007	Enable receive interrupt on completion
IISPI	0x0008 0000	Start of block 0 normal word memory
IMSPI	0x0000 0001	32-bit data transfers
CSPI	0x0000 0180	0x100 instructions = 0x180 32-bit words

From the perspective of the processor, there is no difference between booting from the three types of SPI slave devices. Since SPI is a full-duplex protocol, the processor is receiving the same amount of bits that it sends as

a read command. The read command comprises a full 32-bit word (which is what the processor is initialized to send) comprised of a 24-bit address with an 8-bit opcode. The 32-bit word, received while the read command is transmitted, is thrown away in hardware and can never be recovered by the user. Consequently, special measures must be taken to guarantee that the boot stream is identical in all three cases.

The processor boots in least significant bit first (LSB) format, while most serial memory devices operate in most significant bit first (MSB) format. Therefore, it is necessary to program the device in a fashion that is compatible with the required LSB format. See "Bit-Reverse Option for SPI Boot Modes" on page 6-13 for details.

Also, because the processor always transmits 32 bits before it begins reading boot data from the slave device, the loader utility must insert extra data into the byte stream (in the loader file) if using memory devices that do not use the LSB format. The loader utility includes an option for creating a boot stream compatible with both endian formats, and devices requiring 16-bit and 24-bit addresses, as well as those requiring no read command at all. See "Initial Word Option for SPI Master Boot Modes" on page 6-14 for details.

Figure 6-1 shows the initial 32-bit word sent out from the processor. As shown in the figure, the processor initiates the SPI master boot process by writing an 8-bit opcode (LSB first) to the slave device to specify a read operation. This read opcode is fixed to `0xC0` (`0x03` in MSB first format). Following that, a 24-bit address (all zeros) is always driven by the proces-

sor. On the following SPICLK cycle (cycle 32), the processor expects the first bit of the first word of the boot stream. This transfer continues until the boot kernel has finished loading the user program into the processor.

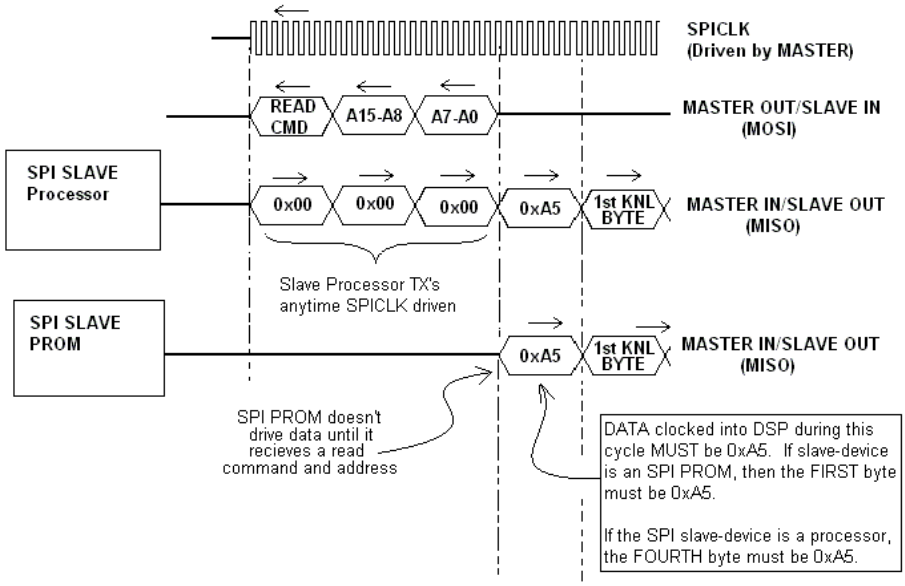
SPICLK
(Driven by MASTER)
READ CMD
A15-A8
A7-A0
MASTER OUT/SLAVE IN (MOSI)
SPI SLAVE Processor
0x00
0x00
0x00
0xA5
1st KNL BYTE
MASTER IN/SLAVE OUT (MISO)
Slave Processor TX's anytime SPICLK driven
SPI SLAVE PROM
0xA5
1st KNL BYTE
MASTER IN/SLAVE OUT (MISO)
SPI PROM doesn't drive data until it recieves a read command and address
DATA clocked into DSP during this cycle MUST be 0xA5. If slave-device is an SPI PROM, then the FIRST byte must be 0xA5.
If the SPI slave-device is a processor, the FOURTH byte must be 0xA5.

Figure 6-1. SPI Master Mode Booting Using Various Serial Devices

Bit-Reverse Option for SPI Boot Modes

SPI PROM. For the SPI PROM boot type, the entirety of the SPI master `.ldr` file needs the option of bit-reversing when loading to SPI PROMs. This is because the default setting for the SPICTL register (see Table 6-8 on page 6-11) sets the bit order to be LSB first. SPI EPROMs are usually MSB first, so the `.ldr` file must be sent in bit-reversed order.

SPI Master and SPI Slave. When loading to other slave devices, the SPI master and SPI slave boot types do not need bit reversing necessarily. For SPI slave and SPI master boots to non-PROM devices, the same default exists (bit-reversed); however, the host (master or slave) can simply be configured to transmit LSB first.

Initial Word Option for SPI Master Boot Modes

Before final formatting (binary, include, etc.) the loader must prepends the word `0xA5` to the beginning of the byte stream. During SPI master booting, the SPI port discards the first byte read from the SPI.

SPI PROM. For the SPI PROM boot type, the word `0xA5` prepended to the stream is one byte in length. SPI PROMs receives a 24-bit read command before any data is sent to the processor, the processor then discards the first byte it receives after this 24-bit opcode is sent (totaling one 32-bit word).

SPI Master. For the SPI master boot type, the word `0xA5000000` prepended to the stream is 32 bits in length. An SPI host configured as a slave begins sending data to the processor while the processor is sending the 24-bit PROM read opcode. These 24-bits must be zero-filled because the processor discards the first 32-bit word that it receives from the slave.

The `0xA5` byte is *only* required for SPI master boot mode.

Figure 6-2 and Table 6-9 illustrates the first 32-bit word for both the SPI PROM and SPI master cases.

With bit reversing for SPI master boot mode, the 32-bit word is handled according to the host width. With bit reversing for SPI PROM boot, the 8-bit word is reversed as a byte and prepended (see Table 6-10).

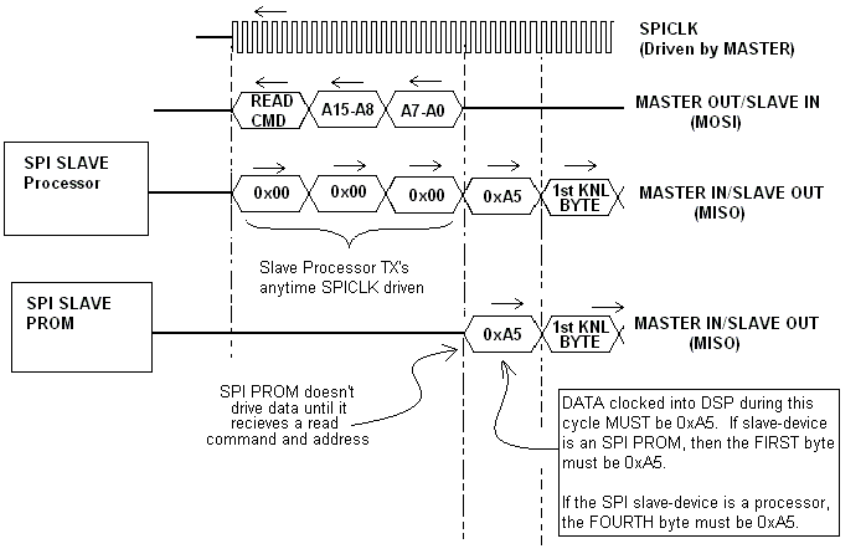

Figure 6-2. SPI Master Boot from a Slave Processor Vs. a Slave PROM

Table 6-9. Initial Word for SPI Master and SPI PROM in .ldr File

Boot Mode	Additional Word	-hostwidth		
		32	16	8
SPI master[1]	0xA5000000	A5000000	0000	00
			A500	00
				00
				A5
SPI PROM[2]	0xA5	A5	A5	A5

1 Initial word for SPI master boot type is always 32 bits. See Figure 6-1 on page 6-13 for explanation.

2 Initial word for SPI PROM boot type is always 8 bits. See Figure 6-1 on page 6-13 for explanation

Table 6-10. Default Settings for PROM and SPI Boot Modes

Boot Type Selection	Host Width	Output Format	Bit Reverse	Initial Word
`-bprom`	8	Intel hex	No	-
`-bspislave`	32	ASCII	No	-
`-bspiflash`	32	ASCII	No	-
`-bspimaster`	32	ASCII	No	`0x000000a5`
`-bspiprom`	8	Intel hex	Yes	`0xa5`

Booting From an SPI Flash

For SPI flash devices, the format of the boot stream is identical to that used in SPI slave mode, with the first byte of the boot stream being the first byte of the kernel. This is because SPI flash devices do not drive out data until they receive an 8-bit command and a 24-bit address.

Booting From an SPI PROM (16-bit address)

Figure 6-2 shows the initial 32-bit word sent out from the processor from the perspective of the serial PROM device.

As shown in Figure 6-2, SPI EEPROMs only require an 8-bit opcode and a 16-bit address. These devices begin transmitting on clock cycle 24. However, because the processor is not expecting data until clock cycle 32, it is necessary for the loader to pad an extra byte to the beginning of the boot stream when programming the PROM. In other words, the first byte of the boot kernel is the second byte of the boot stream. The VisualDSP++ tools automatically handles this in the loader file generation process for SPI PROM devices.

Booting From an SPI Host Processor

Typically, host processors in SPI slave mode transmit data on every `SPICLK` cycle. This means that the first four bytes that are sent by the host processor are part of the first 32-bit word that is thrown away by the processor (see Figure 6-1). Therefore, it is necessary for the loader to pad an extra four bytes to the beginning of the boot stream when programming the host; for example, the first byte of the kernel is the fifth byte of the boot stream. VisualDSP++ automatically handles this in the loader file generation process.

Internal Boot Mode

In internal boot mode, upon reset, the processor starts executing the application stored in the internal boot kernel.

To facilitate internal booting, the `-nokernel` command-line switch commands the loader utility:

- To omit a boot kernel.
  The `-nokernel` switch denotes that a running on the processor (already booted) subroutine imports the `.ldr` file. The loader utility does not insert a boot kernel into the `.ldr` file—a similar subroutine is present already on the processor. Instead, the loader file begins with the first header of the first block of the boot stream.
- To omit any interrupt vector table (IVT) handling.
  In internal boot mode, the boot stream is not imported by a boot kernel executing from within the IVT; no self-modifying `FINAL_INIT` code (which overwrites itself with the IVT) is needed. Thus, the loader utility does not give any special handling to the 256 instructions located in the IVT (`0x80000-0x800FF` for the ADSP-2126x, `0x90000-0x900FF` for the ADSP-2136x, and `0x8C000-0x8C0FF` for the ADSP-2146x processors). Instead, the IVT code or data are handled like any other range of memory.

- To omit an initial word of 0xa5.
  When -nokernel is selected, the loader utility does not place an initial word (A5) in the boot stream as required for SPI master booting.

- To replace the FINAL_INIT block with a USER_MESG header.
  The FINAL_INIT block (which typically contains the IVT code) should not be included in the .ldr file because the contents of the IVT (if any) is incorporated in the boot-stream. Instead, the loader utility appends one final bock header to terminate the loader file.

  The final block header has a block tag of 0x0 (USER_MESG). The header indicates to a subroutine processing the boot stream that this is the end of the stream. The header contains two 32-bit data words, instead of count and address information (unlike the other headers). The words can be used to provide version number, error checking, additional commands, return addresses, or a number of other messages to the importing subroutine on the processor.

  The two 32-bit values can be set on the command line as arguments to the -nokernel[message1, message2] switch. The first optional argument is *msg_word1*, and the second optional argument is *msg_word2*, where the values are interpreted as 32-bit unsigned numbers. If only one argument is issued, that argument is *msg_word1*. It is not possible to specify *msg_word2* without specifying *msg_word*1.) If one or no arguments are issued at the command line, the default values for the arguments are 0x00000000.

  Listing 6-1 shows a sample format for the USER_MESG header.

Listing 6-1. Internal Booting: FINAL_INIT Block Header Format

```
0x00000000    /* USER_MESG tag */
0x00000000    /* msg_word1 (1st cmd-line parameter) */
0x00000000    /* msg_word2 (2nd cmd-line parameter) */
```

ADSP-2126x/2136x/2137x/2146x Processors Boot Kernels

The boot-loading process starts with a transfer of the boot kernel program into the processor memory. The boot kernel sets up the processor and loads boot data. After the boot kernel finishes initializing the rest of the system, the boot kernel loads boot data over itself with a final DMA transfer.

Table 6-11 lists the ADSP-2126x/2136x/2137x/2146x boot kernels shipped with VisualDSP++.

Table 6-11. ADSP-2126x/2136x/2137x/2146x Default Boot Kernel Files

Processor	PROM	SPI Slave, SPI Flash, SPI Master, SPI PROM	Link Port Boot
ADSP-2126x	26x_prom.dxe	26x_spi.dxe	N/A
ADSP-21362, ADSP-21363, ADSP-21364, ADSP-21365, ADSP-21366	36x_prom.dxe	36x_spi.dxe	N/A
ADSP-21367, ADSP-21368, ADSP-21369	369_prom.dxe	369_spi.dxe	N/A
ADSP-2137x	375_prom.dxe	375_spi.dxe	N/A
ADSP-21462, ADSP-21465, ADSP-21467, ADSP-21469	469_prom.dxe	469_spi.dxe	469_link.dxe

At processor reset, a boot kernel is loaded into the `seg_ldr` memory segment as defined in the Linker Description File for the default loader kernel that corresponds to the target processor, for example, `2126x_ldr.ldf`, which is stored in the `<isntall_path>\ldr` VisualDSP++ directory of the target processor.

Boot Kernel Modification and Loader Issues

Boot kernel customization is required for some systems. In addition, the operation of other tools (such as the C/C++ compiler) is influenced by whether the loader utility is used.

If you do not specify a boot kernel file via the **Load** page of the **Project Options** dialog box in VisualDSP++ (or via the `-l` command-line switch), the loader utility places a default boot kernel (see Table 6-11) in the loader output file based on the specified boot type.

If you do not want to use any boot kernel file, check the **No kernel** box (or specify the `-nokernel` command-line switch). The loader utility places no boot kernel in the loader output file.

Rebuilding a Boot Kernel File

If you modify the boot kernel source (`.asm`) file by inserting correct values for your system, you must rebuild the boot kernel (`.dxe`) before generating the boot-loadable (`.ldr`) file. The boot kernel source file contains default values for the `SYSCON` register. The `WAIT`, `SDCTL`, and `SDRDIV` initialization code are in the boot kernel file comments.

To Modify a Boot Kernel Source File

1. Copy the applicable boot kernel source file (`.asm`).
2. Apply the appropriate changes.

After modifying the boot kernel source file, rebuild the boot kernel (`.dxe`) file. Do this from within the VisualDSP++ IDDE (refer to VisualDSP++ online Help for details) or rebuild a boot kernel file from the command line.

Rebuilding a Boot Kernel Using Command Lines

Rebuild a boot kernel using command lines as follows.

PROM Booting. The default boot kernel source file for PROM booting is `26x_prom.asm` for the ADSP-2126x processors. After copying the default file to `my_prom.asm` and modifying it to suit your system, use the following command lines to rebuild the boot kernel.

```
easm21k -proc ADSP-21262 my_prom.asm
linker -T 2162x_ldr.ldf my_prom.doj
```

SPI Booting. The default boot kernel source file for link booting is `2126x_SPI.asm` for the ADSP-2126x processors. After copying the default file to `my_SPI.asm` and modifying it to suit your system, use the following command lines to rebuild the boot kernel:

```
easm21k -proc ADSP-21262 my_SPI.asm
linker -T 2126x_ldr.ldf my_SPI.doj
```

Loader File Issues

If you modify the boot kernel for the PROM or SPI booting modes, ensure that the `seg_ldr` memory segment is defined in the `.ldf` file. Refer to the source of this memory segment in the `.ldf` file located in the `…\ldr` installation directory of the target processor.

Because the loader utility uses the address of `0x80004` (for the ADSP-2126x processors) and the address of `0x9004` (for the ADSP-2136x/2137x/2146x processors) as the first location of the reset vector during the boot-load process, avoid placing code at the address. When using any of the processor's power-up booting modes, ensure that

the address does not contain a critical instruction, because the address is not executed during the booting sequence. Place a NOP or IDLE in this location. The loader utility generates a warning if the vector address 0x80004 for the ADSP-2126x processors (0x90004 for the ADSP-2136x/2137x/2146x processors) does not contain NOP or IDLE.

When using VisualDSP++ to create the loader file, specify the name of the customized boot kernel executable in the **Kernel file** box on the **Load** page of the **Project Options** dialog box.

ADSP-2126x/2136x/2137x/2146x Processors Interrupt Vector Table

If the ADSP-2126x, ADSP-2136x, ADSP-2137x, or ADSP-2146x processor is booted from an external source (PROM or SPI boot modes), the interrupt vector table is located in internal memory (0x80000-0x800FF for the ADSP-2126x processors, 0x90000-0x900FF for the ADSP-2136x/2137x/2146x processors). If the processor is not booted and executes from external memory (no-boot mode), the vector table must be located in external memory.

The IIVT bit in the SYSCTL control register can be used to override the booting mode when determining the location of the interrupt vector table. If the processor is not booted (no-boot mode), setting IIVT to 1 selects an internal vector table, and setting IIVT=0 selects an external vector table. If the processor is booted from an external source (any boot mode other than no-boot), IIVT has no effect. The default initialization value of IIVT is zero.

ADSP-2126x/2136x/2137x/2146x Processor Boot Streams

The loader utility generates and inserts a header at the beginning of a block of contiguous data and instructions in the loader file. The kernel uses headers to properly place blocks into processor memory. The architecture of the header follows the convention used by other SHARC processors.

For all of the ADSP-2126x/2136x/2137x/2146x processor boot types, the structures of block header are the same. The header consists of three 32-bit words: the block tag, word count, and destination address. The order of these words is as follows.

0x000000TT	First word. Tag of the data block (T)
0x0000CCCC	Second word. Data word length or data word count (C) of the data block.
0xAAAAAAAA	Third word. Start address (A) of the data block.

ADSP-2126x/2136x/2137x/2146x Processor Block Tags

Table 6-12 details the ADSP-2126x/2136x/2137x/2146x processor block tags.

Table 6-12. ADSP-2126x/2136x/2137x/2146x Processor Block Tags

Tag	Count[1]	Address	Padding
0x0 FINAL_INIT			None
0x1 ZERO_LDATA	Number of 16-, 32-, or 64-bit words	Logical short, normal, or long word address	None
0x2 ZERO_L48[2]	Number of 48-bit words	Logical normal word address	None

Table 6-12. ADSP-2126x/2136x/2137x/2146x Processor Block Tags

Tag	Count[1]	Address	Padding
0x3 INIT_L16	Number of 16-bit words	Logical short word address	If count is odd, pad with 16-bit zero word; see "INIT_L16 Blocks" on page 6-27 for details.
0x4 INIT_L32	Number of 32-bit words	Logical normal word address	None
0x5 INIT_L48	Number of 48-bit words	Logical normal word address	If count is odd, pad with 48-bit zero word; see "INIT_L48 Blocks" on page 6-26 for details.
0x6 INIT_L64	Number of 64-bit words	Logical long word address	None; see "INIT_L64 Blocks" on page 6-28 for details.
0x7 ZERO_EXT8	Number of 32-bit words	Physical external address	None
0x8 ZERO_EXT16	Number of 32-bit words	Physical external address	None
0x9 INIT_EXT8	Number of 32-bit words	Physical external address	None
0xA INIT_EXT16	Number of 32-bit words	Physical external address	None
0xB MULTI_PROC for ADSP-21367, ADSP-21368, ADSP-21369, ADSP-2137x, ADSP-2146x	Processor IDs (bits 0-7); see on page 6-33 for details.	Offset to the next processor ID in words (32 bits)	None
0x0 USR_MESG	msg_word1	msg_word2	None; see "Internal Boot Mode" on page 6-17 for more info on *msgword*.

1 The count is the actual number of words and does NOT included padded words added by the loader utility.

2 40-bit data and 48-bit words are treated identically.

The ADSP-2126x/2136x/2137x/2146x processor uses eleven block tags, a lesser number of tags compared to other SHARC predecessors. There is only one initialization tag per width because there is no need to draw distinction between `pm` and `dm` sections during initialization. The same tag is used for 16-bit (short word), 32-bit (normal word), and 64-bit (long word) blocks that contain only zeros. The `0x1` tag is used for `ZERO_INIT` blocks of 16-bit, 32-bit, and 64-bit words. The `0x2` tag is used for `ZERO_INIT` blocks of 40-bit data and 48-bit instructions.

For clarity, the letter `L` has been added to the names of the internal block tags. `L` indicates that the associated section header uses the *logical* word count and *logical* address. Previous SHARC boot kernels do not use logical values. For example, the count for a 16-bit block may be the number of 32-bit words rather than the actual number of 16-bit words.

Only four tags are required to handle an external memory, two for each packing mode (see "Packing Options for External Memory" on page 6-7) because parallel port DMA is the only way to access the external memory. The external memory can be accessed only via the *physical* address of the memory. This means that each 32-bit word corresponds to either four (for 8-bit) or two (for 16-bit) external addresses. The `EXT` appended to the name of the block tag indicates that the address is a physical external address. For the ADSP-21367/21368/21369/2137x and ADSP-2146x processors, tag `INIT_L32` also is used for all external 32-bit blocks.

The `0xB` tag is for multiprocessor systems, exclusively supported on the ADSP-21367/21368/21369/2137x and ADSP-2146x processors. The tag indicates that the header is a processor ID header with the ID values and offset values stored in the header. A block can have multiple IDs in its block header, which makes it possible to boot the block into multiple processors.

Two data tags, USER_MESG and FINAL_INIT, differ from the standard format for other SHARC data tags. The USER_MESG header is described on page 6-17 and the FINAL_INIT header on page 6-28.

INIT_L48 Blocks

The INIT_L48 block has one packing and one padding requirements. First, there must be an even number of 48-bit words in the block. If there is an odd number of instructions, then the loader utility must append one additional 48-bit instruction that is all zeros. In all cases, the count placed into the header is the original logical number of words. That is, the count does not include the padded word. Once the number of words in the block is even, the data in this block is packed according to Table 6-13.

Table 6-13. INIT_L48 Block Packing and Zero-Padding (ASCII Format)

Original Data	Packed into an Even Number of 32-bit Words	-hostwidth		
		32	16	8
111122223333	22223333	22223333	3333	33
444455556666	66661111	55551111	2222	33
AAAABBBBCCCC	44445555	44445555	1111	22
	BBBBCCCC	BBBBCCCC	6666	22
	0000AAAA	0000AAAA	5555	11
	00000000	00000000	4444	11
			CCCC	66
			BBBB	66
			AAAA	55
			0000	55
			0000	44
			0000	44
				CC

Table 6-13. INIT_L48 Block Packing and Zero-Padding (ASCII Format)

Original Data	Packed into an Even Number of 32-bit Words	-hostwidth		
		32	16	8
				CC
				BB

INIT_L16 Blocks

For 16-bit initialization blocks, the number of 16-bit words in the block must be even. If an odd number of 16-bit words is in the block, then the loader utility adds one additional word (all zeros) to the end of the block, as shown in Table 6-14. The count stored in the header is the actual number of 16-bit words. The count does not include the padded word.

Table 6-14. INIT_L16 Block Packing and Zero-Padding (ASCII Format)

Original Data	Packed into an Even Number of 32-bit Words	-hostwidth		
		32	16	8
1122	33441122	33441122	1122	22
3344	00005566	00005566	3344	11
5566			5566	44
			0000	33
				66
				55
				00
				00

INIT_L64 Blocks

For 64-bit initialization blocks, the data is packed as shown in Table 6-15.

Table 6-15. INIT_L64 Block Packing (ASCII Format)

Original Data	Packed into an Even Number of 32-bit Words	-hostwidth		
		32	16	8
1111222233334444	33334444	33334444	4444	44
	11112222	11112222	3333	44
			2222	33
			1111	33
				22
				22
				11
				11

FINAL_INIT Blocks

The final 256-instructions of the `.ldr` file contain the instructions for the IVT. The instructions are initialized by a special self-modifying subroutine in the boot kernel (see Listing 6-3). To support the self-modifying code, the loader utility modifies the `FINAL_INIT` block as follows:

1. Places a multi-function instruction at the fifth instruction of the block:
   The loader utility places the instruction `R0=R0-R0, DM(I4,M5)=R9, PM(I12,M13)=R11;` at `0x80004` for the ADSP-2126x processors or

0x90004 for the ADSP-2136x/2137x/2146x processors. The instruction overwrites whatever instruction is at that address. The opcode for this instruction is 0x39732D802000.

2. Places an RTI instruction in the IVT:
   The loader utility places an RTI instruction (opcode 0x0B3E00000000) at the first address in the IVT entry associated with the boot-source, either PROM or SPI. Unlike the multifunction instruction placed at 0x80004 (for the ADSP-2126x processors) or 0x90004 (for the ADSP-2136x/2137x/2146x processors), which overwrites the data, the loader utility preserves the user-specified instruction which the RTI replaces. This instruction is stored in the header for FINAL_INIT as shown in Listing 6-2.
   - For PROM boot mode, the RTI is placed at address 0x80050 for the ADSP-2126x and at 0x90050 for the ADSP-2136x/2137x/2146x processors.
   - For all SPI boot modes, the RTI is placed at address 0x80030 for the ADSP-2126x and at 0x90030 for the ADSP-2136x/2137x/2146x processors (high priority SPI interrupt).
3. Saves an IVT instruction in the FINAL_INIT block header.
   The count and address of a FINAL_INIT block are constant; to avoid any redundancy, the count and address are not placed into the block header. Instead, the 32-bit count and address words are used to hold the instruction that overwrites the RTI inserted into the IVT. Listing 6-1 illustrates the block header for FINAL_INIT if, for example, the opcode 0xAABBCCDDEEFF is assumed to be the user-intended instruction for the IVT.

Listing 6-2. FINAL_INIT Block Header Format

```
0x00000000          /* FINAL_INIT tag = 0x0   */
0xEEFF0000          /* LSBs of instructions   */
0xAABBCCDD          /* 4 MSBs of instructions */
```

Listing 6-3. FINAL_INIT Section

```
/* ====================== FINAL_INIT ======================== */
/* The FINAL_INIT subroutine in the boot kernel program sets up
a DMA to overwrite itself. The code is the very last piece that
runs in the kernel; it is self-modifying code, It uses a DMA
to overwrite itself, initializing the 256 instructions that
reside in the Interrupt Vector Table.                         */
/* ------------------------------------------------------------ */

final_init:

  /* ----------- Setup for IVT instruction patch ------------- */
  I8=0x80030;      /* Point to SPI vector to patch from PX     */
  R9=0xb16b0000;   /* Load opcode for "PM(0,I8)=PX" into R9    */
  PX=pm(0x80002);  /* User instruction destined for 0x80030
                      is passed in the section-header for
                      FINAL_INIT. That instr. is initialized upon
                      completion of this DMA (see comments below)
                      using the PX register. */
  R11=BSET R11 BY 9;  /* Set IMDW to 1 for inst. write    */
  DM(SYSCTL)=R11;     /* Set IMDW to 1 for inst. write    */

  /* ------ Setup loop for self-modifying instruction ------- */
  I4=0x80004;          /* Point to 0x080004 for self-modifying
                          code inserted by the loader at 0x80004
                          in bootstream                       */
  R9=pass R9, R11=R12; /* Clear AZ, copy power-on value
```

```
                          of SYSCTL to R11                    */
   DO 0x80004 UNTIL EQ; /* Set bottom-of-loop address (loopstack)
                          to 0x80004 and top-of-loop (PC Stack)
                          to the address of the next
                          instruction.                        */
   PCSTK=0x80004;       /* Change top-of-loop value from the
                          address of this instruction to
                          0x80004.                            */

   /* ------------- Setup final DMA parameters --------------- */
   R1=0x80000;DM(IISX)=R1; /* Setup DMA to load over ldr       */
   R2=0x180; DM(CSX)=R2;   /* Load internal count              */
   DM(IMSX)=M6;            /* Set to increment internal ptr    */

   /*----------------- Enable SPI interrupt -------------------*/
   bit clr IRPTL SPIHI; /* Clear any pending SPI interr. latch  */
   bit set IMASK SPIHI;   /* Enable SPI receive interrupt       */
   bit set MODE1 IRPTEN;  /* Enable global interrupts           */

   FLUSH CACHE;         /* Remove any kernel instr's from cache */

   /*---------- Begin final DMA to overwrite this code -------- */
   ustat1=dm(SPIDMAC);
   bit set ustat1 SPIDEN;
   dm(SPIDMAC)=ustat1;    /* Begin final DMA transfer          */

   /*------------ Initiate self-modifying sequence ----------- */
   JUMP 0x80004 (DB);    /* Causes 0x80004 to be the return
                            address when this DMA completes and
                            the RTI at 0x80030 is executed.    */
     IDLE;               /* After IDLE, patch then start       */
     IMASK=0;            /* Clear IMASK on way to 0x80004      */

/* ========================================================== */
```

```
/* When this final DMA completes, the high-priority SPI interrupt
is latched, which triggers the following chain of events:

1) The IDLE in the delayed branch to completes
2) IMASK is cleared
3) The PC (now 0x80004 due to the "JUMP RESET (db)") is pushed
   on the PC stack and the processor vectors to 0x80030 to
   service the interrupt.
   Meanwhile, the loader (anticipating this sequence) has auto-
   matically inserted an "RTI" instruction at 0x80030. The user
   instruction intended for that address is instead placed
   in the FINAL_INIT section-header and has loaded into PX before
   the DMA was initiated.)

4) The processor executes the RTI at 0x80030 and vectors to the
   address stored on the PC stack (0x80004).
   Again, the loader has inserted an instruction into the boot
   stream and has placed it at 0x40005 (opcode x39732D802000):
     R0=R0-R0,DM(I4,M5)=R9,PM(I12,M13)=R11;

   This instruction does the following.
   A) Restores the power-up value of SYSCTL (held in R11).
   B) Overwrites itself with the instruction "PM(0,I8)=PX;"
      The first instruction of FINAL_INIT places the opcode for
      this new instruction, 0xB16B00000000, into R9.
   C) R0=R0-R0 causes the AZ flag to be set.

   This satisfies the termination-condition of the loop set up
   in FINAL_INIT ("DO RESET UNTIL EQ;"). When a loop condition
   is achieved within the last three instructions of a loop,
   the processor branches to the top-of-loop address (PCSTK)
   one final time.
```

```
5) We manually changed this top-of-loop address 0x80004, and so
   to conclude the kernel, the processor executes the instruction
   at 0x80004 *again*.

6) There's a new instruction at 0x80004: "PM(0,I8)=PX;". This
   initializes the user-intended instruction at 0x80030 (the vec-
   tor for the High-Priority-SPI interrupt).

At this point, the kernel is finished, and execution continues
at 0x80005, with the only trace as if nothing happened!       */
/* ========================================================== */
```

ADSP-2136x/2137x/2146x Multi-Application (Multi-DXE) Management

Up to eight ADSP-21367/21368/21389/2137x and ADSP-2146x processors can be clustered together and supported by the VisualDSP++ loader utility. In PROM boot mode, all of the processors can boot from the same PROM. The loader utility assigns an input executable (`.dxe`) file to a processor ID or to a number of processor IDs, provided a corresponding loader option is selected on the property page or on the command line. The loader utility inserts the ID into the output boot stream using the multiprocessor tag `MULTI_PROC` (see Table 6-12). The loader utility also inserts the offset (the 32-bit word count of the boot stream built from the input executable (`.dxe`) file) into the boot stream. The `MULTI_PROC` tag enables the boot kernel to identify each section of the boot stream with the executable (`.dxe`) file from which that section was built.

Figure 6-4 shows the multiprocessor boot stream structure.

The processor ID of the corresponding processor is indicated in a 32-bit word, which has the *`Nth`* bit set for the `.dxe` file corresponding to `ID=`*`N`*. Table 6-16 shows all possible ID fields.

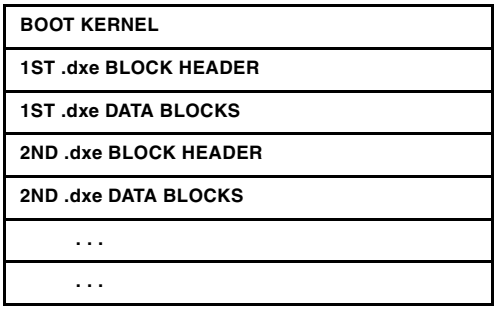

Figure 6-3. Multiprocessor Boot Stream

Table 6-16. Multiprocessor ID Fields

Processor ID Number	Loader ID Field
0	0x00000001
1	0x00000002
2	0x00000004
3	0x00000008
4	0x00000010
5	0x00000020
6	0x00000040
7	0x00000080
1 && 4	0x00000012
6 && 7	0x000000C0

The multiprocessor tag, processor ID, and the offset are encapsulated in a multiprocessor header. The multiprocessor header includes three 32-bit words: the multiprocessor tag; the ID (0-7) of the associated processor .dxe file in the lowest byte of a word; and the offset to the next multiprocessor tag. The loader -id#exe=filename switch is used to assign a

processor ID number to an executable file. The loader -id#ref=N switch is used to share the same executable file by setting multiple bits in the ID field. Figure 6-4 shows the multiprocessor header structure.

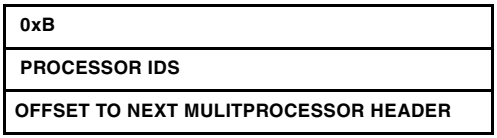

Figure 6-4. Multiprocessor Header

ADSP-2126x/2136x/2137x/2146x Processors Compression Support

The loader utility for the ADSP-2126x/2136x/2137x/2146x processors offers a loader file (boot stream) compression mechanism known as zLib. The zLib compression is supported by a third party dynamic link library, `zLib1.dll`. Additional information about the library can be obtained from the `http://www.zlib.net` Web site.

The zLib1 dynamic link library is included in VisualDSP++. The library functions perform the boot stream compression and decompression procedures when the appropriate options are selected for the loader utility.

The boot kernel with built-in decompression mechanism must perform the decompression on the compressed boot stream in a booting process. The default boot kernel with decompression functions are included in VisualDSP++.

The loader -compression switch directs the loader utility to perform the boot stream compression from the command line. VisualDSP++ also offers a dedicated loader property page (**Load Compression**) to manage the compression from the graphical user interface.

The loader utility takes two steps to compress a boot stream. First, the utility generates the boot stream in the conventional way (builds data blocks), then applies the compression to the boot stream. The decompression initialization is the reversed process: the loader utility decompresses the compressed stream first, then loads code and data into memory segments in the conventional way.

The loader utility compresses the boot stream on the `.dxe`-by-`.dxe` basis. For each input `.dxe` file, the utility compresses the code and data together, including all code and data from any associated shared memory (`.sm`) files. The loader utility, however, does not compress automatically any data from any associated overlay files. To compress data and code from the overlay file, call the utility with the -compressionOverlay switch, either from the property page or from the command line.

Compressed Streams

The basic structure of a loader file with compressed streams is shown in Figure 6-5.

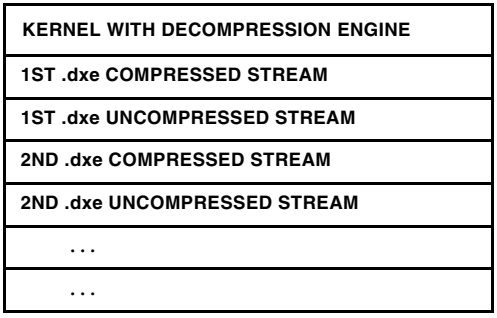

Figure 6-5. Loader File with Compressed Streams

The kernel code with the decompression engine is on the top of the loader file. This section is loaded into the processor first and is executed first when a boot process starts. Once the kernel code is executed, the rest of

the stream is brought into the processor. The kernel code calls the decompression routine to perform the decompression operation on the stream, and then loads the decompressed stream into the processor's memory in the same manner a conventional kernel does when it encounters a compressed stream.

Figure 6-6 shows the structure of a compressed boot stream.

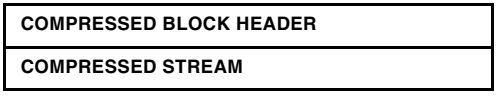

Figure 6-6. Compressed Block

Compressed Block Headers

A compressed stream always has a header, followed by the payload compressed stream.

The compressed block header is comprised of three 32-bit words. The structure of a compressed block header is shows in Figure 6-7.

0X00002000	COMPRESSION TAG/FLAG
0XWBIT0PAD	WINDOW SIZE/PADDED WORD COUNT
0XBYTEBYTE	COMPRESSED BYTE COUNT

Figure 6-7. Compressed Block Header

The first 32-bit word of the compressed block header holds the compression flag, `0x00002000`, which indicates that it is a compressed block header.

The second 32-bit word of the compressed block header hold the size of the compression window (takes the upper 16 bits) and padded word count (takes the lower 16 bits). For the ADSP-2126x/2136x/2137x/2146x processors, the loader utility always rounds the byte count of the compressed stream to be a multiple of 4. The loader utility also pads 3 bytes to the compressed stream if the byte count of the compressed stream from the loader compression engine is not a multiple of 4. An actual padded byte count is a value between `0x0000` and `0x0003`.

The compression window size is 8–15 bits, with the default value of 9 bits. The compression window size specifies to the compression engine a number of bytes taken from the window during the compression. The window size is the 2's exponential value.

The next 32 bits of the compressed block header holds the value of the compressed stream byte count, excluding the byte padded.

A window size selection affects, more or less, the outcome of the data compression. Streams in decompression windows of different sizes are, in general, different and most likely not compatible to each other. If you are building a custom decompression kernel, ensure the same compression window size is used for both the loader utility and the kernel. In general, a bigger compression window size leads to a smaller outcome stream. However, the benefit of a big window size is marginal in some cases. An outcome of the data compression depends on a number of factors, and a compression window size selection is only one of them. The other important factor is the coding structure of an input stream. A compression window size selection can not cause a much smaller outcome stream if the compression ability of the input stream is low.

Uncompressed Streams

Following the compressed streams, the loader utility file includes the uncompressed streams. The uncompressed streams include application codes, conflicted with the code in the initialization blocks in the processor's memory spaces, and a final block. The uncompressed stream includes

only a final block if there is no conflicted code. The final block can have a zero byte count. The final block indicates the end of the application to the initialization code.

Overlay Compression

The loader utility compresses the code and data from the executable `.dxe` and shared memory `.sm` files when the -compression command-line switch is used alone, and leaves the code and data from the overlay (`.ovl`) files uncompressed. The -compressionOverlay switch directs the loader utility to compress the code and data from the `.ovl` files, in addition to compressing the code and data from the `.dxe` and `.sm` files.

The -compressionOverlay switch must be used in conjunction with -compression.

Booting Compressed Streams

Figure 6-8 shows the booting sequence of a loader file with compressed streams. The loader file is pre-stored in the flash memory.

1. A a booting process is initialized by the processor.
2. The processor brings the 256 words of the boot kernel from the flash memory to the processor's memory for execution.
3. The decompression engine is brought in.
4. The compressed stream is brought in, then decompressed and loaded into the memory.
5. The uncompressed stream is brought and loaded into memory, possibly to overwrite the memory spaces taken by the compressed code.
6. The final block is brought and loaded into the memory to overwrite the memory spaces taken by the boot kernel.

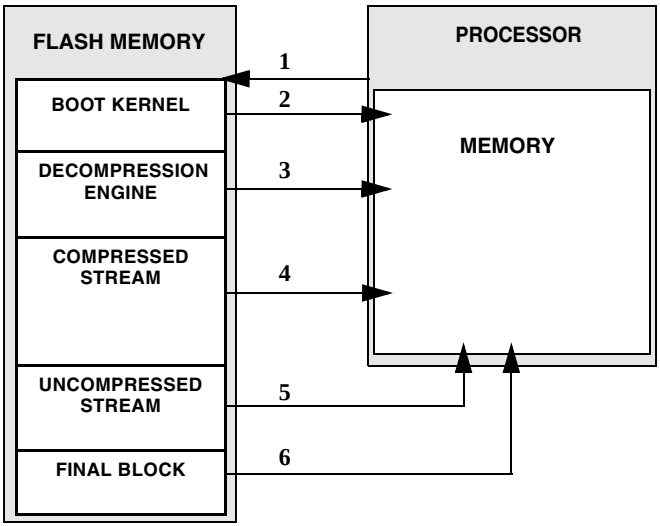

Figure 6-8. ADSP-2126x/2136x/2137x/2146x Compressed Loader Stream: Booting Sequence

Decompression Kernel File

As stated before, a decompression kernel `.dxe` file must be used when building a loader file with compressed streams. The decompression kernel file has a built-in decompression engine to decompress the compressed streams from the loader file.

A decompression kernel file can be specified from the loader property page or from the command line via the -l userkernel switch. VisualDSP++ includes the default decompression kernel files, which the loader utility uses if no other kernel file is specified. If building a custom decompression kernel, ensure that you use the same decompression function, and use the same compression window size for both the kernel and the loader utility.

The default decompression kernel files are stored in the `<install_path>\2126x\ldr\zlib` and `\2136x\ldr\zlib` directories of VisualDSP++. The loader utility uses the window size of 9 bits to perform the compression operation. The compression window size can be changed through the loader property page or the -compressWS # command-line switch. The valid range for the window size is from 8 to 15 bits.

ADSP-2126x/2136x/2137x/2146x Processor Loader Guide

Loader operations depend on the loader options, which control how the loader utility processes executable files. You select features such as boot modes, boot kernels, and output file formats via the loader options. These options are specified on the loader utility's command line or via the **Load** page of the **Project Options** dialog box in the VisualDSP++ environment.

The **Load** page consists of multiple panes. For information specific to the ADSP-2126x/2136x/2137x/2146x processor, refer to the VisualDSP++ online help for that processor. When you open the **Load** page, the default loader settings for the selected processor are already set. Use the **Additional Options** box to enter options that have no dialog box equivalent.

Option settings on the **Load** page correspond to switches displayed on the command line.

These sections describe how to produce a bootable loader file (`.ldr`):

- "Using ADSP-2126x/2136x/2137x/2146x Loader Command Line" on page 6-42
- "Using VisualDSP++ Interface (Load Page)" on page 6-49

Using ADSP-2126x/2136x/2137x/2146x Loader Command Line

Use the following syntax for the SHARC loader command line.

```
elfloader inputfile -proc processor -switch [-switch …]
```

where:

- *inputfile*—Name of the executable file (`.dxe`) to be processed into a single boot-loadable file. An input file name can include the drive and directory. Enclose long file names within straight quotes, "`long file name`".
- `-proc` *processor*—Part number of the processor (for example, `-proc ADSP-21262`) for which the loadable file is built. The `-proc` switch is mandatory.
- `-switch` …—One or more optional switches to process. Switches select operations and boot modes for the loader utility. A list of all switches and their descriptions appear in Table 6-18 on page 6-44.

Command-line switches are not case-sensitive and may be placed on the command line in any order.

The following command line,

```
elfloader Input.dxe -bSPIflash -proc ADSP-21262
```

runs the loader utility with:

- `Input.dxe`—Identifies the executable file to process into a boot-loadable file. Note that the absence of the `-o` switch causes the output file name to default to `Input.ldr`.

- `-bspiflash`—Specifies SPI flash port booting as the boot type for the boot-loadable file.
- `-proc ADSP-21262` —Specifies ADSP-21262 as the target processor.

File Searches

File searches are important in loader processing. The loader utility supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur as described on page 1-17.

File Extensions

Some loader switches take a file name as an optional parameter. Table 6-17 lists the expected file types, names, and extensions.

Table 6-17. File Extensions

Extension	File Description
`.dxe`	Executable files and boot kernel files. The loader utility recognizes overlay memory files (`.ovl`) and shared memory files (`.sm`), but does not expect these files on the command line. Place `.ovl` and `.sm` files in the same directory as the `.dxe` file that refers to them. The loader utility finds the files when processing the `.dxe` file. The `.ovl` and `.sm` files may also be placed in the `.ovl` and `.sm` file output directory specified in the `.ldf` file.
`.ldr`	Loader output file

Loader Command-Line Switches

Table 6-18 is a summary of the ADSP-2126x, ADSP-2136x, ADSP-2137x, and ADSP-2146x loader switches.

Table 6-18. ADSP-2126x/2136x/2137x/2146x Loader Command-Line Switches

Switch	Description

`-bprom`
`-bspislave|-bspi`
`-bspimaster`
`-bspiprom`
`-bspiflash`
	Specifies the boot mode. The `-b` switch directs the loader utility to prepare a boot-loadable file for the specified boot mode.
	The valid modes (boot types) are PROM, SPI slave, SPI master, SPI PROM, and SPI flash.
	If `-b` does not appear on the command line, the default is `-bprom`.
	To use a custom boot kernel, the boot type selected with the `-b` switch must correspond with the boot kernel selected with the `-l` switch. Otherwise, the loader utility automatically selects a default boot kernel based on the selected boot type
	(see "ADSP-2126x/2136x/2137x/2146x Processors Boot Kernels" on page 6-19). Do not use with the `-nokernel` switch.

`-compression`	Directs the loader utility to compress the application data and code, including all data and code from the application-associated shared memory files (see "ADSP-2126x/2136x/2137x/2146x Processors Compression Support" on page 6-35). The data and code from the overlay files are not compressed if this switch is used alone (see `-compressionOverlay`).
`-compressionOverlay`	Directs the loader utility to compress the application data and code from the associated overlay files (see "Overlay Compression" on page 6-39). This switch must be used with `-compression`.
`-compressWS #`	The `-compressWS #` switch specifies a compression window size in bytes. The number is a 2's exponential value to be used by the compression engine. The valid values are `[8-15]`, with the default of `9`.

Table 6-18. ADSP-2126x/2136x/2137x/2146x Loader Command-Line Switches (Cont'd)

Switch	Description

`-fhex`
`-fASCII`
`-fbinary`
`-finclude`
`-fs1`
`-fs2`
`-fs3`
	Specifies the format of a boot-loadable file (Intel hex-32, ASCII, binary, include). If the `-f` switch does not appear on the command line, the default boot file format is
	Intel hex-32 for PROM and SPI PROM, ASCII for SPI slave, SPI flash, and SPI master.
	Available formats depend on the boot type selection (`-b` switch):
	• For PROM and SPI PROM boot types, select a hex, ASCII, s1, s2, s3, or include format.
	• For other SPI boot types, select an ASCII or binary format.

`-h`
or
`-help`
	Invokes the command-line help, outputs a list of command-line switches to standard output, and exits.
	By default, the `-h` switch alone provides help for the loader driver. To obtain a help screen for the target processor, add the `-proc` switch to the command line. For example: type `elfloader -proc ADSP-21262 -h` to obtain help for the ADSP-2126x/2136x and ADSP-2137x processors.

`-hostwidth #`	Sets up the word width for the `.ldr` file. By default, the word width for PROM and SPI PROM boot modes is 8; for SPI slave, SPI flash, and SPI master boot modes is 32. The valid word widths are: • 8 for Intel hex 32 and Motorola S-records formats; • 8, 16, or 32 for ASCII, binary, and include formats
`-id#exe=`*`filename`*	Specifies the processor ID. Directs the loader utility to use the processor ID (#) for a corresponding executable file (the *`filename`* parameter) when producing a boot-loadable file. This switch is used to produce a boot-loadable file to boot multiple processors. Valid values for # are `0`, `1`, `2`, `3`, `4`, `5`, `6`, and `7`. Do not use this switch for single-processor systems. For single-processor systems, use *`filename`* as a parameter without a switch. This switch applies to the ADSP-21367/21368/21369/2137x and ADSP-2146x processors only.

Table 6-18. ADSP-2126x/2136x/2137x/2146x Loader Command-Line Switches (Cont'd)

Switch	Description
`-id#ref=N`	Directs the loader utility to share the boot stream for processor *N* with processor #. If the executable file of the # processor is identical to the executable of the *N* processor, the switch can be used to set the start address of the processor with ID of # to be the same as that of the processor with ID of *N*. This effectively reduces the size of the loader file by providing a single copy of the file to two or more processors in a multiprocessor system. This switch applies to the ADSP-21367/21368/21369/2137x and ADSP-2146x processors only.
`-l userkernel`	Directs the loader utility to use the specified *userkernel* and to ignore the default boot kernel for the boot-loading routine in the output boot-loadable file. **Note**: The boot kernel file selected with this switch must correspond to the boot type selected with the `-b` switch). If the `-l` switch does not appear on the command line, the loader utility searches for a default boot kernel file in the installation directory, (see "ADSP-2126x/2136x/2137x/2146x Processors Boot Kernels" on page 6-19). For kernels with the decompression engine, see "Decompression Kernel File" on page 6-40. The loader utility does not search for any kernel file if `-nokernel` is selected.
`-nokernel[message1, message2]`	Supports internal boot mode. The `-nokernel` switch directs the loader utility: • Not to include the boot kernel code into the loader (`.ldr`) file. • Not to perform any special handling for the 256 instructions located in the IVT. • To put two 32-bit hex messages in the final block header (optional). • Not to include the initial word in the loader file. For more information, see "Internal Boot Mode" on page 6-17.

Table 6-18. ADSP-2126x/2136x/2137x/2146x Loader Command-Line Switches (Cont'd)

Switch	Description
`-o filename`	Directs the loader utility to use the specified *filename* as the name for the loader's output file. If the `-o` *filename* is absent, the default name is the root name of the input file with an `.ldr` extension.
`-noZeroBlock`	The `-noZeroBlock` switch directs the loader utility not to build zero blocks.
`-p`*address*	Specifies the PROM start address. This EPROM address corresponds to `0x80000` (ADSP-2126x processors) or to `0x90000` (ADSP-2136x/2137x/2146x processors). The `-p` switch starts the boot-loadable file at the specified address in the EPROM. If the `-p` switch does not appear on the command line, the loader utility starts the EPROM file at address `0x0`.

`-proc` *processor*	Specifies the processor. This is a mandatory switch. The *processor* argument is one of the following:
	`ADSP-21261` `ADSP-21262` `ADSP-21266`
	`ADSP-21267` `ADSP-21362` `ADSP-21363`
	`ADSP-21264` `ADSP-21365` `ADSP-21366`
	`ADSP-21267` `ADSP-21368` `ADSP-21369`
	`ADSP-21371` `ADSP-21375` `ADSP-21462`
	`ADSP-21465` `ADSP-21467` `ADSP-21469`

`-retainSecondStageKernel`	Directs the loader utility to retain the decompression code in the memory at runtime. The `-retainSecondStageKernel` switch must be used with `-compression`.

Table 6-18. ADSP-2126x/2136x/2137x/2146x Loader Command-Line Switches (Cont'd)

Switch	Description
`-si-revision #|none|any`	The `-si-revision {#|none|any}` switch provides a silicon revision of the specified processor. The switch parameter represents a silicon revision of the processor specified by the `-proc` *processor* switch. The parameter takes one of three forms: • The `none` value indicates that the VisualDSP++ ignores silicon errata. • The # value indicates one or more decimal digits, followed by a point, followed by one or two decimal digits. Examples of revisions are: `0.0`; `1.12`; `23.1`. Revision `0.1` is distinct from and "lower" than revision `0.10`. The digits to the left of the point specify the chip tapeout number; the digits to the right of the point identify the metal mask revision number. The number to the right of the point cannot exceed decimal `255`. • The `any` value indicates that VisualDSP++ produces an output file that can be run at any silicon revision. The switch generates either a warning about any potential anomalous conditions or an error if any anomalous conditions occur. In the absence of the switch parameter (a valid revision value)—`-si-revision` alone or with an invalid value—the loader utility generates an error.
`-v`	Outputs verbose loader messages and status information as the loader utility processes files.
`-version`	Directs the loader utility to show its version information. Type `elfloader -version` to display the version of the loader drive. Add the `-proc` switch, for example, `elfloader -proc ADSP-21262 -version` to display version information of both loader drive and SHARC loader.

Using VisualDSP++ Interface (Load Page)

After selecting a **Loader file** as the target type on the **Project** page in VisualDSP++ **Project Options** dialog box, modify the default options on the **Load** pages (also called loader property page). Click **OK** to save the selections. Selecting **Build Project** from the **Project** menu generates a loader file. For information relative to a specific processor, refer to the VisualDSP++ online help for that processor.

VisualDSP++ invokes the `elfloader` utility to build the output file. Dialog box buttons and fields correspond to command-line switches and parameters (see Table 6-18 on page 6-44). Use the **Additional Options** box to enter options that have no dialog box equivalent.

7 LOADER FOR TIGERSHARC PROCESSORS

This chapter explains how the loader utility (`elfloader.exe`) is used to convert executable (`.dxe`) files into boot-loadable or non-bootable files for the ADSP-TSxxx TigerSHARC processors.

Refer to "Introduction" on page 1-1 for the loader utility's overview; the introductory material applies to all processor families. Loader operations specific to the ADSP-TSxxx TigerSHARC processors are detailed in the following sections.

- "TigerSHARC Processor Booting" on page 7-2
  Provides general information on various booting modes, including information on boot kernels.
- "TigerSHARC Loader Guide" on page 7-5
  Provides reference information on the loader utility's command-line syntax and switches.

Refer to the processor's data sheet and hardware reference manual for more information on system configuration, peripherals, registers, and operating modes.

TigerSHARC Processor Booting

At chip reset, a TigerSHARC processor loads (bootstraps) a 256-instruction program (called a boot kernel) into the processor's internal memory. The boot kernel program may be stored on an external PROM, a host processor, or another TigerSHARC processor. The boot type is selected via the processor's boot mode select ($\overline{\text{BMS}}$) pin as described in "Boot Type Selection" on page 7-3. After the boot kernel loads, it executes itself and then loads the rest of the application program and data into the processor. The combination of the boot kernel and the application program comprises a boot-loadable file.

TigerSHARC processors support three booting modes: EPROM/flash, host, and link. The boot-loadable files for each of these modes pack the boot data into 32-bit instructions and use a DMA channel of the processor's DMA controller to boot-load the instructions.

Additionally, there are several no-boot modes, which do not require kernels.

- In EPROM/flash boot mode, the loader utility generates a PROM image that contains all project data and loader code. The project data is then stored in an 8-bit wide external EPROM. After reset, the processor performs a special booting scenario, reading the EPROM content through the processor's external port and initializing on-chip and off-chip memories.
- In host boot mode, the processor accepts boot data from a 32- or 64-bit synchronous microprocessor (host). The host writes a boot-loadable file to the processor's `AUTODMA` register through the processor's external port, one 32-bit word at a time. Once the last word is written, the processor takes over and runs the user code.

- In link port boot mode, the processor receives boot data via its link port from another TigerSHARC processor.

EE-174: ADSP-TS101S TigerSHARC Processor Boot Loader Kernels Operation and *EE-200: ADSP-TS20x TigerSHARC Processor Boot Loader Kernels Operation* provide additional information about the loader. These EE notes are available from the Analog Devices Web site at: http://www.analog.com/processors/processors/tigersharc/technicalLibrary/index.html.

Boot Type Selection

To determine the boot mode, a TigerSHARC processor samples its boot mode select ($\overline{\text{BMS}}$) pin. While the processor is held in reset, the $\overline{\text{BMS}}$ pin is an active input.

If $\overline{\text{BMS}}$ is sampled low a certain number of clock cycles after reset, EPROM/flash boot is selected and, after RESET goes high, $\overline{\text{BMS}}$ becomes an output, acting as EPROM chip select.

If $\overline{\text{BMS}}$ is sampled high after reset, the TigerSHARC processor is at an IDLE state, waiting for a host or link boot.

The 100K Ohm internal pull-down on $\overline{\text{BMS}}$ may not suffice, depending on the line loading. Thus, an additional external pull-down resistor may be necessary for the EPROM boot mode. If host or link boot is desired, $\overline{\text{BMS}}$ must be high and may be tied directly to the system power bus.

TigerSHARC Processor Boot Kernels

Upon completion of the DMA, in all boot modes, the boot-loading process continues by downloading the boot kernel into the processor memory. The boot kernel sets up and initializes the processor's memory. After initializing the rest of the system, the boot kernel overwrites itself.

You can build an `.ldr` file that includes or does not include a kernel. To build without a kernel, use the `-nokernel` command-line switch or uncheck the **Use boot kernel** option on the **Kernel** page of the **Project Options** dialog box.

VisualDSP++ includes three distinct kernel programs for each TigerSHARC processor. A boot kernel is loaded at reset into a memory segment, `seg_ldr`, which is defined in the `ADSP-TSxxx_Loader.ldf` file. The provided files are located in the `<isntall_path>\TS\ldf` directory.

Table 7-1. TigerSHARC Boot Kernel Source Files

PROM Boot Kernel	Host Boot Kernel	Link Port Boot Kernel
Ts101_prom.asm	Ts101_host.asm	Ts101_link.asm
Ts201_prom.asm	Ts201_host.asm	Ts201_link.asm
Ts202_prom.asm	Ts202_host.asm	Ts202_link.asm
Ts203_prom.asm	Ts203_host.asm	Ts203_link.asm

Boot Kernel Modification

For most systems, some customization of the boot kernel is required. The operation of other tools (notably the C/C++ compiler) is influenced by loader usage.

For more information on boot kernel operations, refer to the comments in the corresponding boot kernel source files and application notes *EE-174: ADSP-TS101S TigerSHARC Processor Boot Loader Kernels Operation* and *EE-200: ADSP-TS20x TigerSHARC Processor Boot Loader Kernels Operation*. The notes can be found at: `http://www.analog.com/processors/processors/tigersharc/technicalLibrary/index.html`.

TigerSHARC Loader Guide

Loader operations depend on the loader options, which control how the loader utility processes executable files. You select features such as boot modes, boot kernels, and output file formats via the loader options. These options are specified on the loader utility's command line or via the **Load** page of the **Project Options** dialog box in the VisualDSP++ environment. When you open the **Load** page, the default loader settings for the selected processor are already set.

Option settings on the **Load** page correspond to switches displayed on the command line.

These sections describe how to produce a bootable file (`.ldr`):

- "Using TigerSHARC Loader Command Line" on page 7-6
- "Using VisualDSP++ Interface (Load Page)" on page 7-12

Using TigerSHARC Loader Command Line

The TigerSHARC loader utility uses the following command-line syntax.

For a single input file:

```
elfloader inputfile -proc processor [-switch …]
```

For multiple input files:

```
elfloader id1exe=inputfile.dxe id2exe=inputfile2.dxe… -proc processor [-switch …]
```

where:

- *inputfile*—Name of the executable file (`.dxe`) to be processed into a single boot-loadable. An input file name can include the drive and directory.

  For multiprocessor or multi-input systems, specify multiple input `.dxe` files. Use the `-id#exe=` switch, where `#` is the ID number (from `0` to `7`) of the processor. Enclose long file names within straight quotes, "`long file name`".

- `-proc` *processor*—Part number of the processor (for example, `ADSP-TS101`) for which the loadable file is built.

- *-switch …*—One or more optional switches to process. Switches select operations and modes for the loader utility.

Command-line switches may be placed on the command line in any order. For a multi-input system, the loader utility processes the input executable files in the ascending order from the `-id#exe=` switch presented on the command line.

```
elfloader p0.dxe -proc ADSP-TS101 -bprom -fhex -l Ts101_prom.dxe
```

In the above example, the command line runs the loader utility with:

- `p0.dxe`—Identifies the executable file to process into a boot-loadable file. Note the absence of the `-o` switch causes the output file name to default to `p0.ldr`.
- `-proc ADSP-TS101`—Specifies ADSP-TS101 as the processor type.
- `-bprom`—Specifies EPROM booting as the boot type for the boot-loadable file.
- `-fhex`—Specifies Intel hex-32 format for the boot-loadable file.
- `-l TS101_prom.exe`—Specifies the boot kernel file to be used for the boot-loadable file.

```
elfloader -id2exe=p0.dxe -id3exe=p1.dxe -proc ADSP-TS101 -bprom
  -fhex -l Ts101_prom.dxe
```

In the above example, the command line runs the loader utility with:

- `p0.dxe`—Identifies the executable file for the processor with ID of `2` to process into a boot-loadable file. Note the absence of the `-o` switch causes the output file name to default to `p0.ldr`.
- `p1.dxe`—Identifies the executable file for the processor with ID of `3` to process into a boot-loadable file.
- `-proc ADSP-TS101`—Specifies ADSP-TS101 as the processor type.
- `-bprom`—Specifies EPROM booting as the boot type for the boot-loadable file.
- `-fhex`—Specifies Intel hex-32 format for the boot-loadable file.
- `-l Ts101_prom.exe`—Specifies the boot kernel file to be used for the boot-loadable file.

File Searches

File searches are important in loader processing. The loader utility supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur as described on page 1-1.

File Extensions

Some loader switches take a file name as an optional parameter. Table 7-2 lists the expected file types, names, and extensions. The loader utility takes files with extensions of `.dxe`, `.ovl`, and `.sm` but expects only those with extension `.dxe` in a command line on the **Load** page. The loader utility finds files with extensions of `.ovl` and `.sm` as it processes the associated `.dxe` file. The loader utility searches for `.ovl` and `.sm` files in the directory holding the `.dxe` files, in the directory specified in the `.ldf` file, or in the current directory.

Table 7-2. TigerSHARC File Extensions

Extension	File Description
`.dxe`	Loader input files and boot kernel files
`.ldr`	Loader output file
`.ovl`	Overlay files. The loader utility does not expect them on a command line
`.sm`	Shared memory files. The loader utility does not expect them on a command line

TigerSHARC Command-Line Switches

A summary of the loader command-line switches appears in Table 7-3.

Table 7-3. TigerSHARC Loader Command-Line Switches

Switch	Description

`-bprom`
`-bhost`
`-blink`
	Prepares a boot-loadable file for the specified boot mode. Valid boot types include PROM, host, and link port. If the `-b` switch does not appear on the command line, the default setting is `-bprom`. To use a custom kernel, the boot type selected with the `-b` switch must correspond to the boot kernel selected with the `-l` switch.

`-fhex`
`-fASCII`
`-fbinary`
`-fs1`
`-fs2`
`-fs3`
	Prepares a boot-loadable file in the specified format. Available format selections are: hex (Intel hex-32), s1, s2, s3 (Motorola S-records), include, ASCII, and binary. Valid formats depend on the `-b` switch boot type selection.
	• For a PROM boot type, use a hex, s1, s2, s3, include, binary, or ASCII format.
	• For host or link port booting, use ASCII or binary formats.
	If the `-f` switch does not appear on the command line, the default boot type format is hex for PROM, and ASCII for host or link.

`-h`
or
`-help`
	Invokes the command-line help, outputs a list of command-line switches to standard output, and exits. By default, the `-h` switch alone provides help for the loader driver. To obtain a help screen for the target TigerSHARC processor, add the `-proc` switch to the command line. For example, type `elfloader -proc ADSP-TS101 -h` to obtain help for the ADSP-TS101S processor.

`-id#exe=`*`filename`*	Directs the loader utility to use the processor ID number for the corresponding executable file when producing a boot-loadable file for a EPROM- or host-boot multiprocessor system. Use this switch only to produce a boot-loadable file that boots multiple processors from a single EPROM. Valid # are `0`, `1`, `2`, `3`, `4`, `5`, `6`, and `7`. **Warning:** Do not use this switch for single-processor systems. For single-processor systems, use the executable file name as a parameter without a switch.

Table 7-3. TigerSHARC Loader Command-Line Switches (Cont'd)

Switch	Description
`-l userkernele`	Directs the loader utility to use the specified *`userkernel`* and to ignore the default boot kernel for the boot-loading routine in the output boot-loadable file. **Note:** The boot kernel file selected with this switch must correspond to the boot type selected with the `-b` switch). If `-l` does not appear on the command line, the loader utility searches for a default boot kernel file in the installation directory (see "TigerSHARC Processor Boot Kernels" on page 7-4).
`-nokernel`	Supports internal boot mode. The `-nokernel` switch directs the loader utility not to include the boot kernel code into the loader (`.ldr`) file.
`-o filename`	Directs the loader utility to use the specified *`filename`* as the name of the loader output file. If the *`filename`* is absent, the default name is the name of the input file with an `.ldr` extension.
`-p #`	Specifies the EPROM start address (hex format) for the boot-loadable file. If the `-p` switch does not appear on the command line, the loader utility starts the EPROM file at address `0x0` in the EPROM; this EPROM address corresponds to address `0x4000000` in a TigerSHARC processor.
`-proc processor`	Specifies the target processor. The *`processor`* can be one of the following: `ADSP-TS101`, `ADSP-TS201`, `ADSP-TS202`, or `ADSP-TS203`.
`-t #`	Sets the number of timeout cycles (#) as a maximum number of cycles the processor spends initializing external memory. Valid values range from `3` to `32765` cycles; `32765` is the default value. The timeout value is directly related to the number of cycles the processor locks the bus for boot-loading, instructing the processor to lock the bus for no more than 2x timeout number of cycles. When working with a fast host that cannot tolerate being locked out of the bus, use a relatively small timeout value.
`-v`	Outputs verbose loader messages and status information as the loader utility processes files.

Table 7-3. TigerSHARC Loader Command-Line Switches (Cont'd)

Switch	Description
`-version`	Directs the loader utility to display its version information. Type `elfloader -version` to display the version of the loader drive. Add the `-proc` switch, such as in `elfloader -proc ADSP-TS201 -version` to display version information for the loader drive and TigerSHARC loader utility.
`-si-revision #|none|any`	The `-si-revision {#|none|any}` switch provides a silicon revision of the specified processor. The switch parameter represents a silicon revision of the processor specified by the `-proc` *processor* switch. The parameter takes one of three forms: • The `none` value indicates that the VisualDSP++ ignores silicon errata. • The `#` value indicates one or more decimal digits, followed by a point, followed by one or two decimal digits. Examples of revisions are: `0.0`; `1.12`; `23.1`. Revision `0.1` is distinct from and "lower" than revision `0.10`. The digits to the left of the point specify the chip tapeout number; the digits to the right of the point identify the metal mask revision number. The number to the right of the point cannot exceed decimal `255`. • The `any` value indicates that VisualDSP++ produces an output file that can be run at any silicon revision. The switch generates either a warning about any potential anomalous conditions or an error if any anomalous conditions occur. In the absence of the silicon revision switch, the loader utility selects the greatest silicon revision it is aware of, if any. In the absence of the switch parameter (a valid revision value)—`-si-revision` alone or with an invalid value—the loader utility generates an error.

Using VisualDSP++ Interface (Load Page)

After selecting a **Loader file** as the target type on the **Project** page in VisualDSP++ **Project Options** dialog box, modify the default options on the **Load** page (also called loader property page). Click **OK** to save the selections. Selecting **Build Project** from the **Project** menu generates a loader file. For information relative to a specific processor, refer to the VisualDSP++ online help for that processor.

VisualDSP++ invokes the `elfloader` utility to build the output file. Dialog box buttons and fields correspond to command-line switches and parameters (see Table 7-3 on page 7-9). Use the **Additional Options** box to enter options that have no dialog box equivalent.

8 SPLITTER FOR SHARC AND TIGERSHARC PROCESSORS

This chapter explains how the splitter utility (`elfspl21k.exe`) is used to convert executable (`.dxe`) files into non-bootable files for the ADSP-21xxx SHARC and ADSP-TSxxx TigerSHARC processors. Non-bootable PROM image files execute from external memory of a processor. For TigerSHARC processors, the splitter utility creates a 32-bit image file. For SHARC processors, the utility creates a 64-/48-/40-/32-bit image file or an image file to match a physical memory size.

For SHARC processors, the splitter utility also properly packs the external memory data or code to match the specified external memory widths if the logical width of the data or code is different from that of the physical memory.

In most instances, developers working with SHARC and TigerSHARC processor use the loader utility instead of the splitter. One of the exceptions is a SHARC system that can execute instructions from external memory. The non-bootable PROM image files are often used with the ADSP-21065L processor systems, which have limited internal memory.

Refer to "Introduction" on page 1-1 for the splitter utility overview; the introductory material applies to both processor families.

Splitter operations are detailed in the following sections.

- "Splitter Command Line" on page 8-2
  Provides reference information about the splitter utility's command-line syntax and switches.
- "VisualDSP++ Interface (Split Page)" on page 8-9
  Provides reference information about the splitter utility's graphical user interface.

Splitter Command Line

Use the following syntax for the SHARC and TigerSHARC splitter command line.

```
elfspl21k [-switch …] -pm &|-dm &|-64 &| -proc part_number inputfile
```

or

```
elfspl21k [-switch …] -s section_name inputfile
```

where:

- *inputfile*—Specifies the name of the executable file (`.dxe`) to be processed into a non-bootable file for a single-processor system. The name of the *inputfile* file must appear at the end of the command. The name can include the drive, directory, file name, and file extension. Enclose long file names within straight quotes; for example, "`long file name`".
- *-switch …*—One or more optional switches to process. Switches select operations for the splitter utility. Switches may be used in any order. A list of the splitter switches and their descriptions appear in Table 8-2 on page 8-5.

- `-pm &| -dm &| -64`—For SHARC processors, the `&|` symbol between the switches indicates `AND/OR`. The splitter command line must include one or more of `-pm`, `-dm`, or `-64` (or the `-s` switch). The `-64` switch corresponds to `DATA64` memory space.

TigerSHARC processors do not have `-pm`, `-dm`, or `-64` switches.

- `-s` *`section_name`*—The `-s` switch can be used without the `-pm`, `-dm`, or `-64` switch. The splitter command line must include one or more of the `-pm`, `-dm`, and, `-64` switches or the `-s` switch.

Most items in the splitter command line are not case sensitive; for example, `-pm` and `-PM` are interchangeable. However, the names of memory sections must be identical, including case, to the names used in the executable.

Each of the following command lines,

```
elfspl21k -pm -o pm_stuff my_proj.dxe -proc ADSP21161
elfspl21k -dm -o dm_stuff my_proj.dxe -proc ADSP21161
elfspl21k -64 -o 64_stuff my_proj.dxe -proc ADSP21161
elfspl21k -s seg-code -o seg-code my_proj.dxe
```

runs the splitter utility for the ADSP-21161 processor. The first command produces a PROM file for program memory. The second command produces a PROM file for data memory. The third command produces a PROM file for `DATA64` memory. The fourth command produces a PROM file for section `seg-code`.

The switches on these command lines are as follows.

File Searches

File searches are important in the splitter process. The splitter utility supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur as described on page 1-17.

`-pm`
`-dm`
`-64`
	Selects program memory (`-pm`), data memory (`-dm`), or `DATA64` memory (`-64`) as sources in the executable for extraction and placement into the image. `DATA64` memory does not apply to the ADSP-2106x processors.
	Warning: The `-pm`, `-dm`, or `-64` switch does not apply to the ADSP-TSxxx processors.
	Because these are the only switches used to identify the memory source, the specified sources are `PM`, `DM`, or `DATA64` memory sections. Because no other content switches appear on these command lines, the output file format defaults to a Motorola 32-bit format, and the PROM word width of the output defaults to 8 bits for all PROMs.

`-o pm_stuff`
`-o dm_stuff`
`-o seg-code`
	Specify names for the output files. Use different names so the output of a run does not overwrite the output of a previous run. The output names are `pm_stuff.s_#` and `dm_stuff.s_#`. The splitter utility adds the `.s_#` file extension to the output files; # is a number that differentiates one output file from another.

`my_proj.dxe`	Specifies the name of the input (`.dxe`) file to be processed into non-bootable PROM image files.

Output File Extensions

The splitter utility follows the conventions shown in Table 8-1 for output file extensions.

Table 8-1. Output File Extensions

Extension	File Description
`.s_#`	Motorola S-record format file. The # indicates the position (`0` = least significant, `1` = next-to-least significant, and so on). For info about Motorola S-record file format, refer to "Output Files in Motorola S-Record Format" on page A-10.
`.h_#`	Intel hex-32 format file. The # indicates the position (`0` = least significant, `1` = next-to-least significant, and so on). For information about Intel hex-32 file format, refer to "Splitter Output Files in Intel Hex-32 Format" on page A-12.
`.stk`	Byte-stacked format file. These files are intended for host transfer of data, not for PROMs. For more information about byte stacked file format, format files, refer to "Splitter Output Files in Byte-Stacked Format" on page A-12.

Splitter Command-Line Switches

A list of the splitter command-line switches appears in Table 8-2.

Table 8-2. Splitter Command-Line Switches

Item	Description
`-64`	The `-64` (include `DATA64` memory) switch directs the splitter utility to extract all sections declared as 64-bit memory sections from the input `.dxe` file. The switch influences the operation of the `-ram` and `-norom` switches, adding 64-bit data memory as their target.
`-dm`	The `-dm` (include data memory) switch directs the splitter utility to extract memory sections declared as data memory ROM from the input `.dxe` file. The `-dm` switch influences the operation of the `-ram` and `-norom` switches, adding data memory as their target.
`-o` *`imagefile`*	The `-o` (output file) switch directs the splitter utility to use *`imagefile`* as the name of the splitter output file(s). If not specified, the default name for the splitter output file is `inputfile.ext`, where `ext` depends on the output format.
`-norom`	The `-norom` (no ROM in PROM) switch directs the splitter utility to ignore ROM memory sections in the *`inputfile`* when extracting information for the output image. The `-dm` and `-pm` switches select data memory or program memory. The operation of the `-s` switch is not influenced by the `-norom` switch.
`-pm`	The `-pm` (include program memory) switch directs the splitter utility to extract memory sections declared program memory ROM from the input `.dxe` file. The `-pm` switch influences the operation of the `-ram` and `-norom` switches, adding program memory as the target.

Table 8-2. Splitter Command-Line Switches (Cont'd)

Item	Description
`-r # [# …]`	The `-r` (PROM widths) switch specifies the number of PROM files and their width in bits. The splitter utility can create PROM files for 8-, 16-, and 32-bit wide PROMs. The default width is 8 bits. Each # parameter specifies the width of one PROM file. Place # parameters in order from most significant to least significant. The sum of the # parameters must equal the bit width of the destination memory (40 bits for DM, 48 bits for PM, or 64 bits for 64-bit memory). **Example:** `elfspl21k -dm -r 16 16 8 myfile.dxe` This command extracts data memory ROM from `myfile.dxe` and creates the following output PROM files. • `myfile.s_0`—8 bits wide, contains bits 7–0 • `myfile.s_1`—16 bits wide, contains bits 23–8 • `myfile.s_2`—16 bits wide, contains bits 39–24 The width of the three output files is 40 bits.
`-ram`	The `-ram` (include RAM in PROM) switch directs the splitter utility to extract RAM sections from the *`inputfile`*. The `-dm`, `-pm`, and `-64` switches select the memory. The `-s` switch is not influenced by the `-ram` switch.

`-f h`
`-f s1`
`-f s2`
`-f s3`
`-f b`
	The `-f` (PROM file format) switch directs the splitter utility to generate a non-bootable PROM image file in the specified format.
	Available selection include:
	• `h`—Intel hex-32 format
	• `s1`—Motorola EXORciser format
	• `s2`—Motorola EXORMAX format
	• `s3`—Motorola 32-bit format
	• `b`—byte stacked format
	If the `-f` switch does not appear on the command line, the default format for the PROM file is Motorola 32-bit (`s3`).
	For information on file formats, see "Build Files" on page A-4.

`-s` *`section_name`*	The `-s` (include memory section) switch directs the splitter utility to extract the contents of the specified memory section (*`section_name`*). Use the `-s` *`section_name`* switch as many times as needed. Each instance of the `-s` switch can specify only one *`section_name`*. **Warning:** Do not use `-s` with (`-pm`, `-dm`, or `-64`).

Table 8-2. Splitter Command-Line Switches (Cont'd)

Item	Description
`-proc` *part_number*	Specifies the processor type to the splitter utility. This is a mandatory switch. Valid processors are: • ADSP-21060, ADSP-21061, ADSP-21062, ADSP-21065L • ADSP-21160, ADSP-21161 • ADSP-21261, ADSP-21262, ADSP-21266, ADSP-21267, • ADSP-21363, ADSP-21364, ADSP-21365, ADSP-21366, ADSP-21367, ADSP-21368, ADSP-21369, • ADSP-21371, ADSP-21375 • ADSP-TS101, ADSP-TS201, ADSP-TS202, and ADSP-TS203
`-u` *#*	(Byte-stacked format files only) The `-u` (user flags) switch, which may be used only in combination with the `-f b` switch, directs the splitter utility to use the number *#* in the user-flags field of a byte stacked format file. If the `-u` switch is not used, the default value for the number is `0`. By default, *#* is decimal. If *#* is prefixed with `0x`, the splitter utility interprets the number as hexadecimal. For more information, see "Splitter Output Files in Byte-Stacked Format" on page A-12.

Table 8-2. Splitter Command-Line Switches (Cont'd)

Item	Description
`-si-revision #|none|any`	The `-si-revision {#|none|any}` switch provides a silicon revision of the specified processor. The switch parameter represents a silicon revision of the processor specified by the `-proc` *`processor`* switch. The parameter takes one of three forms: • The `none` value indicates that the VisualDSP++ ignores silicon errata. • The *#* value indicates one or more decimal digits, followed by a point, followed by one or two decimal digits. Examples of revisions are: `0.0`; `1.12`; `23.1`. Revision `0.1` is distinct from and "lower" than revision `0.10`. The digits to the left of the point specify the chip tapeout number; the digits to the right of the point identify the metal mask revision number. The number to the right of the point cannot exceed decimal `255`. • The `any` value indicates that VisualDSP++ produces an output file that can be run at any silicon revision. The switch generates either a warning about any potential anomalous conditions or an error if any anomalous conditions occur. In the absence of the silicon revision switch, the loader selects the greatest silicon revision it is aware of, if any. In the absence of the switch parameter (a valid revision value)—`-si-revision` alone or with an invalid value—the loader generates an error.
`-version`	Directs the splitter utility to show its version information.

VisualDSP++ Interface (Split Page)

After selecting a **Splitter file** as the target type on the **Project** page in VisualDSP++ **Project Options** dialog box, modify the default options on the **Project: Split** page (also called splitter property page). Click **OK** to save the selections. Selecting **Build Project** from the **Project** menu invokes the splitter utility to build a non-bootable PROM image file.

Splitter operation relies on splitter options, which control the processing of the executable files into output files. The page buttons and fields correspond to the splitter utility's command-line switches and parameters (see Table 8-2 on page 8-5). Use the **Additional Options** box to enter options that do not have dialog box equivalents. Refer to VisualDSP++ online Help for details.

A FILE FORMATS

VisualDSP++ development tools support many file formats, in some cases several for each development tool. This appendix describes file formats that are prepared as inputs and produced as outputs.

The appendix describes three types of files:

- "Source Files" on page A-1
- "Build Files" on page A-4
- "Debugger Files" on page A-15

Most of the development tools use industry-standard file formats. These formats are described in "Format References" on page A-16.

Source Files

This section describes the following source (input) file formats.

- "C/C++ Source Files" on page A-2
- "Assembly Source Files" on page A-2
- "Assembly Initialization Data Files" on page A-2
- "Header Files" on page A-3
- "Linker Description Files" on page A-4
- "Linker Command-Line Files" on page A-4

C/C++ Source Files

C/C++ source files are text files (`.c`, `.cpp`, `.cxx`, and so on) containing C/C++ code, compiler directives, possibly a mixture of assembly code and directives, and, typically, preprocessor commands.

Several dialects of C code are supported: pure (portable) ANSI C, and at least two subtypes[1] of ANSI C with ADI extensions. These extensions include memory type designations for certain data objects, and segment directives used by the linker to structure and place executable files.

The C/C++ compiler, run-time library, as well as a definition of ADI extensions to ANSI C, are detailed in the *VisualDSP++ 5.0 C/C++ Compiler and Library Manual for Blackfin Processors*.

Assembly Source Files

Assembly source files (`.asm`) are text files containing assembly instructions, assembler directives, and (optionally) preprocessor commands. For information on assembly instructions, see the Programming Reference manual for your processor.

The processor's instruction set is supplemented with assembly directives. Preprocessor commands control macro processing and conditional assembly or compilation.

For information on the assembler and preprocessor, see the *VisualDSP++ 5.0 Assembler and Preprocessor Manual.*

Assembly Initialization Data Files

Assembly initialization data files (`.dat`) are text files that contain fixed- or floating-point data. These files provide initialization data for an assembler `.VAR` directive or serve in other tool operations.

[1] With and without built-in function support; a minimal differentiator. There are others dialects.

When a .VAR directive uses a .dat file for data initialization, the assembler reads the data file and initializes the buffer in the output object file (.doj). Data files have one data value per line and may have any number of lines.

The .dat extension is explanatory or mnemonic. A directive to #include <*filename*> can take any file name and extension as an argument.

Fixed-point values (integers) in data files may be signed, and they may be decimal, hexadecimal, octal, or binary based values. The assembler uses the prefix conventions listed in Table A-1 to distinguish between numeric formats.

For all numeric bases, the assembler uses words of different sizes for data storage. The word size varies by the processor family,

Table A-1. Numeric Formats

Convention	Description
0x*number* H#*number* h#*number*	Hexadecimal number
number D#*number* d#*number*	Decimal number
B#*number* b#*number*	Binary number
O#*number* o#*number*	Octal number

Header Files

Header files (.h) are ASCII text files that contain macros or other preprocessor commands which the preprocessor substitutes into source files. For information on macros and other preprocessor commands, see the *VisualDSP++ 5.0 Assembler and Preprocessor Manual.*

Linker Description Files

Linker description files (`.ldf`) are ASCII text files that contain commands for the linker in the linker scripting language. For information on the scripting language, see the *VisualDSP++ 5.0 Linker and Utilities Manual*.

Linker Command-Line Files

Linker command-line files (`.txt`) are ASCII text files that contain command-line inputs for the linker. For more information on the linker command line, see the *VisualDSP++ 5.0 Linker and Utilities Manual*.

Build Files

Build files are produced by VisualDSP++ development tools while building a project. This section describes the following build file formats.

- "Assembler Object Files" on page A-5
- "Library Files" on page A-5
- "Linker Output Files" on page A-5
- "Memory Map Files" on page A-6
- "Loader Output Files in Intel Hex-32 Format" on page A-6
- "Loader Output Files in Include Format" on page A-8
- "Loader Output Files in Binary Format" on page A-9
- "Output Files in Motorola S-Record Format" on page A-10
- "Splitter Output Files in Intel Hex-32 Format" on page A-12

- "Splitter Output Files in Byte-Stacked Format" on page A-12
- "Splitter Output Files in ASCII Format" on page A-14

Assembler Object Files

Assembler output object files (`.doj`) are binary object and linkable files (ELF). Object files contain relocatable code and debugging information for a DSP program's memory segments. The linker processes object files into an executable file (`.dxe`). For information on the object file's ELF format, see "Format References" on page A-16.

Library Files

Library files (`.dlb`), the output of the archiver, are binary, object and linkable files (ELF). Library files (called archive files in previous software releases) contain one or more object files (archive elements).

The linker searches through library files for library members used by the code. For information on the ELF format used for executable files, refer to "Format References" on page A-16.

The archiver automatically converts legacy input objects from COFF to ELF format.

Linker Output Files

The linker's output files (`.dxe`, `.sm`, `.ovl`) are binary executable files (ELF). The executable files contain program code and debugging information. The linker fully resolves addresses in executable files. For information on the ELF format used for executable files, see the TIS Committee texts cited in "Format References" on page A-16.

The loaders/splitter utilities are used to convert executable files into boot-loadable or non-bootable files.

Executable files are converted into a boot-loadable file (`.ldr`) for the ADI processors using a splitter utility. Once an application program is fully debugged, it is ready to be converted into a boot-loadable file.
A boot-loadable file is transported into and run from a processor's internal memory. This file is then programmed (burned) into an external memory device within your target system.

A splitter utility generates non-bootable, PROM-image files by processing executable files and producing an output PROM file. A non-bootable, PROM-image file executes from processor external memory.

Memory Map Files

The linker can output memory map files (`.xml`), which are ASCII text files that contain memory and symbol information for the executable files. The `.xml` file contains a summary of memory defined with the `MEMORY{}` command in the `.ldf` file, and provides a list of the absolute addresses of all symbols.

Loader Output Files in Intel Hex-32 Format

The loader utility can output Intel hex-32 format files (`.ldr`). The files support 8-bit-wide PROMs and are used with an industry-standard PROM programmer to program memory devices. One file contains data for the whole series of memory chips to be programmed.

The following example shows how the Intel hex-32 format appears in the loader's output file. Each line in the Intel hex-32 file contains an extended linear address record, a data record, or the end-of-file record.

`:020000040000FA`	Extended linear address record
`:0402100000FE03F0F9`	Data record
`:00000001FF`	End-of-file record

Extended linear address records are used because data records have a 4-character (16-bit) address field, but in many cases, the required PROM size is greater than or equal to 0xFFFF bytes. Extended linear address records specify bits 31–16 for the data records that follow.

Table A-2 shows an example of an extended linear address record.

Table A-2. Extended Linear Address Record Example

Field	Purpose
:020000040000FA	Example record
:	Start character
02	Byte count (always 02)
0000	Address (always 0000)
04	Record type
0000	Offset address
FA	Checksum

Table A-3 shows the organization of a sample data record.

Table A-3. Data Record Example

Field	Purpose
:0402100000FE03F0F9	Example record
:	Start character
04	Byte count of this record
0210	Address
00	Record type
00	First data byte
F0	Last data byte
F9	Checksum

Table A-4 shows an end-of-file record.

Table A-4. End-of-File Record Example

Field	Purpose
:00000001FF	End-of-file record
:	Start character
00	Byte count (zero for this record)
0000	Address of first byte
01	Record type
FF	Checksum

VisualDSP++ includes a utility program to convert an Intel hexadecimal file to Motorola S-record or data file. Refer to "hexutil – Hex-32 to S-Record File Converter" on page B-2 for details.

Loader Output Files in Include Format

The loader utility can output include format files (`.ldr`). These files permit the inclusion of the loader file in a C program.

The word width (8- or16-bit) of the loader file depends on the specified boot type. Similar to Intel hex-32 output, the loader output in include format have some basic parts in the following order.

1. Initialization code (some Blackfin processors)
2. Boot kernel (some Blackfin, SHARC, and TigerSHARC processors)
3. User application code

4. Saved user code in conflict with the initialization code (some Blackfin processors)
5. Saved user code in conflict with the kernel code (some Blackfin, SHARC, and TigerSHARC processors)

The initialization code is an optional first part for some Blackfin processors, while the kernel code is the part for some Blackfin, SHARC, and TigerSHARC processors. User application code is followed by the saved user code.

Files in include format are ASCII text files that consist of 48-bit instructions, one per line (on SHARC processors). Each instruction is presented as three 16-bit hexadecimal numbers. For each 48-bit instruction, the data order is lower, middle, and then upper 16 bits. Example lines from an include format file are:

```
0x005c, 0x0620, 0x0620,
0x0045, 0x1103, 0x1103,
0x00c2, 0x06be, 0x06be
```

This example shows how to include this file in a C program:

```
const unsigned loader_file[] =
   {
   #include “foo.ldr”
   };
const unsigned loader_file_count = sizeof loader_file
                                 / sizeof loader_file[0];
```

The `loader_file_count` reflects the actual number of elements in the array and cannot be used to process the data.

Loader Output Files in Binary Format

The loader utility can output binary format files (`.ldr`) to support a variety of PROM and microcontroller storage applications.

Binary format files use less space than the other loader file formats. Binary files have the same contents as the corresponding ASCII file, but in binary format.

Output Files in Motorola S-Record Format

The loader and splitter utilities can output Motorola S-record format files (.s_#), which conform to the Intel standard. The three file formats supported by the loader and PROM splitter utilities differ only in the width of the address field: S1 (16 bits), S2 (24 bits), or S3 (32 bits).

An S-record file begins with a header record and ends with a termination record. Between these two records are data records, one per line:

S00600004844521B	Header record
S10D00043C4034343426142226084C	Data record (S1)
S903000DEF	Termination record (S1)

Table A-5 shows the organization of an example header record.

Table A-5. Header Record Example

Field	Purpose
S00600004844521B	Example record
S0	Start character
06	Byte count of this record
0000	Address of first data byte
484452	Identifies records that follow
1B	Checksum

Table A-6 shows the organization of an S1 data record.

Table A-6. S1 Data Record Example

Field	Purpose
S10D00043C40343434261422260 84C	Example record
S1	Record type
0D	Byte count of this record
0004	Address of the first data byte
3C	First data byte
08	Last data byte
4C	Checksum

The S2 data record has the same format, except that the start character is S2 and the address field is six characters wide. The S3 data record is the same as the S1 data record except that the start character is S3 and the address field is eight characters wide.

Termination records have an address field that is 16-, 24-, or 32 bits wide, whichever matches the format of the preceding records. Table A-7 shows the organization of an S1 termination record.

Table A-7. S1 Termination Record Example

Field	Purpose
S903000DEF	Example record
S9	Start character
03	Byte count of this record
000D	Address
EF	Checksum

The S2 termination record has the same format, except that the start character is S8 and the address field is six characters wide.

The S3 termination record is the same as the S1 format, except the start character is S7 and the address field is eight characters wide.

For more information, see "hexutil – Hex-32 to S-Record File Converter" on page B-2.

Splitter Output Files in Intel Hex-32 Format

The splitter utility can output Intel hex-32 format (.h_#) files. These ASCII files support a variety of PROM devices. For an example of how the Intel hex-32 format appears for an 8-bit wide PROM, see "Loader Output Files in Intel Hex-32 Format" on page A-6.

The splitter utility prepares a set of PROM files. Each PROM holds a portion of each instruction or data. This configuration differs from the loader output.

Splitter Output Files in Byte-Stacked Format

The splitter utility can output files in byte-stacked (.stk) format. These files are not intended for PROMs, but are ideal for microcontroller data transfers.

A file in byte-stacked format comprises a series of one line headers, each followed by a block (one or more lines) of data. The last line in the file is a header that signals the end of the file.

Lines consist of ASCII text that represents hexadecimal digits. Two characters represent one byte. For example, F3 represents a byte whose decimal value is 243.

Table A-8 shows an example of a header record in byte-stacked format.

In the above example, the start address and block length fields are 32 (0x20) bits wide. The file contains program memory data (the MSB is the only flag currently used in the PROM splitter flags field). No user flags are

Table A-8. Example – Header Record in Byte-Stacked Format

Field	Purpose
2000800000000080000001E	Example record
20	Width of address and length fields (in bits)
00	Reserved
80	PROM splitter flags (80 = PM, 00 = DM)
00	User defined flags (loaded with -u switch)
00000008	Start address of data block
0000001E	Number of bytes that follow

set. The address of the first location in the block is 0x08. The block contains 30 (1E) bytes (5 program memory code words). The number of bytes that follow (until next header record or termination record) must be non-zero.

A block of data records follows its header record, five bytes per line for data memory, and six byte per line for program memory or in other physical memory width. For example:

Program Memory Section (Code or Data)

```
3C4034343426
142226083C15
```

Data Memory Section

```
3C40343434
2614222608
```

DATA64 Memory Section

```
1122334455667788
99AABBCCDDEEFF00
```

The bytes are ordered left to right, most significant to least.

The termination record has the same format as the header record, except for the rightmost field (number of records), which is all zeros.

Splitter Output Files in ASCII Format

When the Blackfin splitter utility is invoked as a splitter utility, its output can be an ASCII format file with the `.ldr` extension. ASCII format files are text representations of ROM memory images that can be post-processed by users.

Data Memory (DM) Example:

```
ext_data { TYPE(DM ROM) START(0x010000) END(0x010003) WIDTH(8) }
```

The above DM section results in the following code.

```
00010000     /* 32-bit logical address field */
00000004     /* 32-bit logical length field */
00020201     /* 32-bit control word: 2x address multiply */
             /* 02 bytes logical width, 01 byte physical width */
00000000     /* reserved */
0x12         /* 1st data word, DM data is 8 bits */
0x56
0x9A
0xDE         /* 4th (last) data word */
CRC16        /* optional, controlled by the -checksum switch */
```

Debugger Files

Debugger files provide input to the debugger to define support for simulation or emulation of your program. The debugger consumes all the executable file types produced by the linker (`.dxe`, `.sm`, `.ovl`). To simulate IO, the debugger also consumes the assembler data file format (`.dat`) and the loadable file formats (`.ldr`).

The standard hexadecimal format for a SPORT data file is one integer value per line. Hexadecimal numbers do not require a `0x` prefix. A value can have any number of digits but is read into the `SPORT` register as follows.

- The hexadecimal number is converted to binary.
- The number of binary bits read in matches the word size set for the `SPORT` register and starts reading from the LSB. The `SPORT` register then zero-fills bits shorter than the word size or conversely truncates bits beyond the word size on the MSB end.

In the following example (Table A-9), a `SPORT` register is set for 20-bit words, and the data file contains hexadecimal numbers. The simulator converts the hex numbers to binary and then fills/truncates to match the `SPORT` word size. The `A5A5` is filled and `123456` is truncated.

Table A-9. SPORT Data File Example

Hex Number	Binary Number	Truncated/Filled
A5A5A	1010 0101 1010 0101 1010	1010 0101 1010 0101 1010
FFFF1	1111 1111 1111 1111 0001	1111 1111 1111 1111 0001
A5A5	1010 0101 1010 0101	0000 1010 0101 1010 0101
5A5A5	0101 1010 0101 1010 0101	0101 1010 0101 1010 0101
11111	0001 0001 0001 0001 0001	0001 0001 0001 0001 0001
123456	0001 0010 0011 0100 0101 0110	0010 0011 0100 0101 0110

Format References

The following texts define industry-standard file formats supported by VisualDSP++.

- Gircys, G.R. (1988) *Understanding and Using COFF* by O'Reilly & Associates, Newton, MA
- (1993) *Executable and Linkable Format (ELF) V1.1* from the Portable Formats Specification V1.1, Tools Interface Standards (TIS) Committee.

  Go to: `http://developer.intel.com/vtune/tis.htm`.
- (1993) *Debugging Information Format (DWARF) V1.1* from the Portable Formats Specification V1.1, UNIX International, Inc.

  Go to: `http://developer.intel.com/vtune/tis.htm`.
- (2001-2005) *uClinux - BFLT Binary Flat Format* by Craig Peacock from the beyondlogic.org.

  Go to: `http://www.beyondlogic.org/uClinux/bflt.htm`.

B UTILITIES

The VisualDSP++ development software includes several utility programs, some of which run from a command line only.

This appendix describes the following utilities.

- "hexutil – Hex-32 to S-Record File Converter" on page B-2
- "elf2flt – ELF to BFLT File Converter" on page B-3
- "fltdump – BFLT File Dumper" on page B-4

Other VisualDSP++ utilities, for example, the ELF file dumper, are described in the *VisualDSP++ 5.0 Linker and Utilities Manual* or online Help.

hexutil – Hex-32 to S-Record File Converter

The hex-to-S file converter (`hexutil.exe`) utility transforms a loader (`.ldr`) file in Intel hexadecimal 32-bit format to Motorola S-record format or produces an unformatted data file.

Syntax: `%hexutil` *`input_file`* `[-s1|s2|s3|StripHex] [-o` *`file_name`*`]`

where:

`input_file` is the name of the `.ldr` file generated by the VisualDSP++ splitter utility.

Table B-1 shows optional switches used with the `%hexutil` command.

Table B-1. Hex to S-Record File Converter Command-Line Switches

Switch	Description
`-s1`	Specifies Motorola output format S1
`-s2`	Specifies Motorola output format S2
`-s3`	Specifies the default output format – Motorola S3. That is, when no switch appears on the command lines, the output file format defaults to S3.
`-StripHex`	Generates an unformatted data file
`-o`	Names the output file; in the absence of the `-o` switch, causes the output file name to default to `input_file.s`.

The Intel hex-32 and Motorola S-record file formats are described on page A-6 and on page A-10, respectively.

elf2flt – ELF to BFLT File Converter

The ELF-to-BFLT file converter (`elf2flt.exe`) utility converts a (`.dxe`) file in Executable and Linkable Format (ELF) to Binary Flat Format (BFLT).

The `.bflt` file contains three output sections: `text`, `data`, and `bss`. Output sections are defined by the ELF file standard. The `.bflt` file can be loaded and executed in an environment running a uClinux operating system.

For more information on the BFLT file format, see uClinux Web site: `http://www.beyondlogic.org/uClinux/bflt.htm`.

The `elf2flt` currently supports ELF files compiled for Blackfin and SHARC architectures. The `elf2flt` implements revision 5 flat relocation type. For more information, see the `BFLT` relocation structure defined in `flat.h`.

`Elf2flt` does not support ELF files with position-independent code and global offset table (`PIC` with `GOT`).

`Elf2flt` is not capable of compressing text and data segments with `gzip` tool.

Syntax: `elf2flt [-V|r|k] [-s #] [-o` *`file_name`*`] elf_input_file`

where:

`elf_input_file` is the name of the `.dxe` file generated by the VisualDSP++ linker.

Table B-2 shows optional switches used with the `elf2flt` command.

Table B-2. ELF to BFLT File Converter Command-Line Switches

Switch	Description
`-V`	Verbose operation
`-r`	Forces load to RAM
`-k`	Enables kernel trace on load (for debug)
`-s#`	Sets application stack-size number
`-o file_name`	Names the output file
`-h`	Prints the list of the `elf2flt` switches
`-v`	Prints version information

fltdump – BFLT File Dumper

The BFLT file dumper (`fltdump.exe`) utility extracts data from BFLT-format executable (`.bflt`) files and yields text showing the BFLT file's contents.

The `fltdump` utility prints the entire contents of the `.bflt` file in hex. In addition, the `fltdump` prints contents of the text section as a list of disassembled machine instructions.

For more information on the BFLT file format, see uClinux Web site: `http://www.beyondlogic.org/uClinux/bflt.htm`.

Syntax: `fltdump [switch…] [object_file]`

where:

`object_file` is the name of the `.bflt` file whose contents is to be printed.

Table B-3 shows optional switches used with the `fltdump` command.

Table B-3. BFLT File Dumper Command-Line Switches

Switch	Description
`-D`	Dumps the file built for the specified processor
`-help`	Prints the list of the `elfdump` switches to `stdout`
`-v`	Prints version information
`-o` *`file_name`*	Prints s the output to the specified file

I INDEX

Numerics

-64 splitter switch, 8-5
16- to 48-bit word packing, 4-12
32- to 16-bit word packing, 6-7
32- to 8-bit word packing, 6-7
48- to 8-bit word packing, 4-9
4- to 48-bit word packing, 4-15
8- to 48-bit word packing, 4-10, 4-11, 4-12, 5-5, 5-9

A

ACK pin, 4-8, 4-11, 4-13, 5-7
ADDR23-0 address lines, 5-8
ADDR31-0 address lines, 4-10
address records, linear format, A-7
ADSP-2106x/160 processors
  ADSP-21060/061/062 boot modes, 4-2, 4-5
  ADSP-21065L boot modes, 4-2, 4-6
  ADSP-21160 boot modes, 4-2, 4-5
  boot sequence, 4-3
  direct memory access, *See* DMA, DMACx
ADSP-21161 processors
  boot modes, 5-2, 5-5
  boot sequence, 5-3
  direct memory access, *See* DMA, DMACx
  multiprocessor support, 5-21
ADSP-2126x/36x/37x/46x processors
  boot modes, 6-2, 6-4, 6-7
  boot sequence, 6-3
  compression support, 6-35
  direct memory access, *See* DMA, DMACx
ADSP-2136x/37x/46x processors
  multiprocessor support, 6-33
ADSP-BF51x processors
  boot modes, 2-3
  multi-dxe loader files, 2-17
ADSP-BF52x/54x processors
  boot modes, 2-4
  multi-dxe loader files, 2-17
ADSP-BF531/2/3/4/6/7/8/9 processors
  ADSP-BF534/6/7 (only) boot modes, 3-4
  boot modes, 3-3
  boot sequence, 3-8
  boot streams, 3-9, 3-10
  compression support, 3-49
  memory ranges, 3-19
  multi-dxe loader files, 3-46
  on-chip boot ROM, 3-3, 3-7, 3-9, 3-19, 3-47
ADSP-BF535 processors
  boot modes, 3-16, 3-21, 3-79
  boot sequence, 3-24
  boot streams, 3-27, 3-28
  memory ranges, 3-33
  on-chip boot ROM, 3-21, 3-23, 3-25
  second stage loader, 3-24

ADSP-BF561 processors
boot modes, 3-35
boot streams, 3-37, 3-40
dual-core architecture, 3-35, 3-38
memory ranges, 3-45
multi-dxe loader files, 3-46
multiprocessor support, 3-44
on-chip boot ROM, 3-35, 3-37, 3-43, 3-44, 3-45, 3-47
.ALIGN directive, 3-34
application loading (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors, 3-9, 3-12, 3-48
ADSP-BF535 processors, 3-26, 3-27, 3-32
ADSP-BF561 processors, 3-37, 3-38, 3-44, 3-45, 3-48
application loading (SHARC processors)
ADSP-2106x/160 processors, 4-4, 4-17
ADSP-21161 processors, 5-3, 5-5, 5-9
ADSP-2126x/36x/37x/46x processors, 6-4, 6-17
ADSP-2126x/36x/37x processors, 6-30
applications
See also blocks of application code
loading, introduction to, 1-15
code start address, 2-14, 2-20, 3-66, 3-74, 4-4, 4-19, 5-5
default code start address, 2-20, 3-74
development flow, 1-7
multiple-dxe files, 2-17
archive files, *See* library files (.dlb)
archiver, A-5
ASCII file format, 2-9, 3-62, 7-9, A-4, A-14
.asm (assembly) source files, 1-8, A-2
assembling, introduction to, 1-8
assembly
directives, A-2
initialization data files (.dat), A-2
object files (.doj), A-5
source text files (.asm), 1-8, A-2
asynchronous FIFO boot mode, ADSP-BF52x/54x processors, 2-4
AUTODMA register, 7-2

B

-baudrate #, loader switch for Blackfin, 3-60
baud rate (Blackfin processors), 3-24, 3-38, 3-74
BFLAG_CALLBACK block flag, 2-9
BFLAG_QUICKBOT block flag, 2-15
BFLAG_SAVE block flag, 2-16
BFLT file dumper, B-4
binary flat format (.bflt), B-3, B-4
binary format files (.ldr), 2-9, 3-62, 7-9, A-9
bit-reverse option (SHARC processors), 6-13
block
of application code, introduction to, 1-16
byte counts (Blackfin processors), 2-13, 3-64
flags, *See* flag words
packing, *See* data packing
tags, 4-18, 5-17, 6-18, 6-23, 6-25
block headers (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors, 3-9, 3-10
ADSP-BF535 processors, 3-33
ADSP-BF561 processors, 3-39, 3-43
block headers (SHARC processors)
ADSP-2106x/160 processors, 4-17
ADSP-21161 processors, 5-17
ADSP-2126x/36x/37x processors, 6-18, 6-23
blocks of application code (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors, 3-9
ADSP-BF535 processors, 3-32
ADSP-BF561 processors, 3-37
blocks of application code (SHARC processors)
ADSP-2106x/160 processors), 4-17
ADSP-21161 processors, 5-17
ADSP-2126x/36x/37x processors, 6-23
BMODE1-0 pins
ADSP-BF531/2/3/8/9 processors, 3-4, 3-16, 3-79

BMODE2-0 pins
ADSP-BF51x processors, 2-3
ADSP-BF534/6/7 processors, 3-4
ADSP-BF535 processors, 3-16, 3-21, 3-79
BMODE3-0 pins
ADSP-BF52x/54x processors, 2-4
BMS pins
ADSP-2106x/160 processors, 4-5, 4-7, 4-10, 4-11, 4-15, 4-23
ADSP-21161 processors, 5-4, 5-6, 5-9, 5-13, 5-15, 5-22
TigerSHARC processors, 7-2, 7-3
boot
sequences, introduction to, 1-11
ROM, *See* on-chip boot ROM
sources, *See* boot modes
BOOT_CFG1-0 pins, 6-4, 6-5, 6-7
BOOT_CFG2-0 pins, 6-5, 6-7
boot differences (Blackfin processors), 3-3, 3-8, 3-35, 3-37, 3-38
boot differences (SHARC processors), 6-11, 6-16
boot file formats
specifying for Blackfin processors, 2-9, 2-20, 3-62, 3-73
specifying for SHARC processors, 4-29, 5-28, 6-45
specifying for TigerSHARC processors, 7-9
boot kernels
See also kernels, second-stage loaders
introduction to, 1-15
boot-loadable files
introduction to, 1-9, 1-10
versus non-bootable file, 1-15
boot modes (Blackfin processors)
ADSP-BF51x processors, 2-3
ADSP-BF52x/54x processors, 2-4
ADSP-BF531/2/3/8/9 processors, 2-3
ADSP-BF534/6/7 processors, 3-4
ADSP-BF535 processors, 3-21
ADSP-BF561 processors, 3-35
specifying, 2-8, 2-19, 3-60, 3-73
boot mode select pins (Blackfin processors)
ADSP-BF51x processors, 2-3
ADSP-BF52x/54x processors, 2-4
ADSP-BF531/2/3/4/6/7/8/9 processors, 3-4
ADSP-BF535 processors, 3-21
boot mode select pins (SHARC processors)
ADSP-21161 processors, 5-4
ADSP-2116x/160 processors, 4-5, 4-6
ADSP-2126x/36x/37x processors, 6-4
ADSP-2146x processors, 6-5
boot mode select pins (TigerSHARC processors), 7-2, 7-3
boot modes (SHARC processors)
ADSP-2106x/160 processors, 4-2, 4-7
ADSP-21161 processors, 5-2, 5-5
ADSP-2126x/36x/37x/46x processors, 6-2
ADSP-2126x/36x/37x processors, 6-4
ADSP-2146x processors, 6-5
specifying, 4-28, 5-28, 6-4, 6-5, 6-44
boot process, introduction to, 1-9
boot sequences (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors, 3-8
ADSP-BF535 processors, 3-8, 3-24
ADSP-BF561 processors, 3-35
boot sequences (SHARC processors)
ADSP-21161 processors, 5-3
ADSP-2116x/160 processors, 4-3
ADSP-2126x/36x/37x/46x processors, 6-3
bootstraps, 1-14, 1-15, 3-65, 7-2
boot streams, introduction to, 1-14, 1-15, 1-16

boot streams (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors, 3-9, 3-47
ADSP-BF535 processors, 3-27, 3-28
ADSP-BF561 processors, 3-37, 3-40, 3-47
similarities between, 3-9
boot streams (SHARC processors)
ADSP-2106x/160 processors, 4-17
ADSP-21161 processors, 5-17
ADSP-2126x/36x/37x processors, 6-17, 6-23
-b prom|flash|spi|spislave|UART|TWI|FIFO, loader switch for ADSP-BF53x processors, 3-60
-b prom|flash|spi|spislave|UART|TWI|FIFO|OTP|NAND, loader switch for ADSP-BF51x/52x/54x processors, 2-8
-bprom|host|link, loader switch for TigerSHARC, 7-9, 7-10
-bprom|host|link|JTAG, loader switch for ADSP-2106x/160 processors, 4-28
-bprom|host|link|spi, loader switch for ADSP-21161 processors, 5-28
-bprom|spislave|spiflash|spimaster|spiprom, loader switch for ADSP-2126x/36x/37x/46x processors, 6-16, 6-44
BSEL pin, 4-6
BSO bit, 4-10
build file formats, list of, A-4
BUSLCK bit, 4-13, 4-14
bypass mode, *See* no-boot mode
byte-stacked format files (.stk), 8-4, 8-6, 8-7, A-12

C

-caddress, loader switch for ADSP-2106x/160 processors, 4-28
-callback, loader switch for Blackfin, 2-9
C and C++ source files, 1-8, A-2
CEP0 register, 4-8, 5-8, 5-9, 5-11, 5-12
CLB0 register, 5-13, 5-14
CLKPL bit, 6-9, 6-11
COFF to ELF file conversion, A-5
command line
loader for SHARC processors, 4-26, 5-25, 6-42
loader for TigerSHARC processors, 7-6
loader/splitter for Blackfin processors, 2-6, 3-58
splitter, 8-2, 8-5
compilation, introduction to, 1-8
compressed block headers
Blackfin processors, 3-12, 3-51
SHARC processors, 6-37
compressed streams
Blackfin processors, 3-50, 3-54
SHARC processors, 6-36, 6-39
-compression
loader switch for Blackfin, 3-49, 3-61
loader switch for SHARC, 6-35, 6-39, 6-44
Compression (Load) page (Blackfin processors), 3-75
-compressionOverlay, loader switch for SHARC, 6-36, 6-39, 6-44
compression support
ADSP-2126x/36x/37x/46x processors, 6-35
ADSP-BF531/2/3/4/6/7/8/9 processors, 3-49, 3-75
compression window, 3-51, 3-56, 6-38, 6-41
-compressWS
loader switch for Blackfin, 3-56, 3-61
loader switch for SHARC, 6-41, 6-44
conversion utilities, B-2
count headers (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors, 3-46
ADSP-BF561 processors, 3-39, 3-43, 3-46
CPEP0 register, 5-8, 5-11
CPHASE bit, 6-9, 6-11
CPLB0 register, 5-14

-CRC32, loader switch for Blackfin, 2-8
CS pin, 4-13, 5-12, 6-8
CSPI register, 6-9, 6-11
CSRX register, 5-16
customer support, -xv
Cx register, 4-8, 4-9, 4-11, 4-15

D

D23-16 bits, 4-10
D39-32 bits, 4-10
D7-0 bits, 4-10
data
  initialization files (.dat), A-2
  memory (dm) sections, 8-3, 8-5
  records in Intel hex-32 format, A-7
  transfers, *See* DMA transfers
DATA15-0 pins, 4-12
DATA23-16 pins, 4-7, 5-6
DATA31-16 pins, 4-12
DATA39-32 pins, 4-7
DATA47-16 pins, 4-12
DATA63-32 pins, 4-12
DATA64 memory sections, 8-4, 8-5
DATA7-0 pins, 4-7, 4-12
data banks (Blackfin processors)
  ADSP-BF531/2/3/4/6/7/8/9 processors, 3-19
  ADSP-BF535 processors, 3-33
  ADSP-BF561 processors, 3-45
DataFlash devices, 3-4
data packing (SHARC processors)
  ADSP-2106x/160 processors, 4-9, 4-10, 4-11, 4-12, 4-15
  ADSP-21161 processors, 5-5, 5-9
  ADSP-2126x/36x/37x/46x processors, 6-7, 6-26, 6-27
data streams
  encrypting from application, 3-61
  encrypting from kernel, 3-63
.dat (data) initialization files, A-2
debugger file formats, 1-8, A-15
debugging targets, 1-9
decompression
  initialization files, 3-54
  kernel files, 6-40
DEN register, 5-7, 5-11
.dlb (library) files, A-5
-dm, splitter switch, 8-5
DMA (ADSP-2106x/160 processors)
  channels, *See* channels by name (DMACx)
  buffers, 4-13
  channel control registers, 4-10, 4-12, 4-13, 4-14, 4-15, 4-16
  channel interrupts, 4-13, 4-14, 4-16
  channel parameter registers, 4-8, 4-9, 4-11, 4-12, 4-16
  controller, 4-2, 4-8, 4-9, 4-11
  transfers, 4-10, 4-11, 4-12, 4-13, 4-16, 4-18
DMA (ADSP-21161 processors)
  channels, *See* channels by name (DMACx)
  buffers, 5-22
  channel control registers, 5-5, 5-6, 5-9, 5-10, 5-11, 5-16
  channel interrupts, 5-9, 5-12
  channel parameter registers, 5-7, 5-9, 5-11, 5-12, 5-13, 5-15, 5-16
  controller, 5-5, 5-7, 5-9
  transfers, 5-3, 5-8, 5-15, 5-16, 5-21
DMA (ADSP-2126x/36x/37x/46x processors)
  code example, 6-30
  parallel port channels, 6-6, 6-25
  parameter registers, 6-6, 6-8, 6-11, 6-31
  SPI channels, 6-9, 6-11
  transfers, 6-6, 6-8, 6-10, 6-19
DMAC0 channel (ADSP-2106x/160 processors), 4-3, 4-8, 4-12
DMAC10 channels
  ADSP-2106x/160 processors, 4-2, 4-3, 4-8, 4-9, 4-12, 4-15
  ADSP-21161 processors, 5-5, 5-6, 5-7, 5-9, 5-10

DMAC6 channel (ADSP-2106x/160 processors), 4-2, 4-3, 4-8, 4-9, 4-12, 4-15
DMAC8 channels
  ADSP-2106x/160 processors, 4-2, 4-3, 4-12, 4-15
  ADSP-21161 processors, 5-2, 5-12, 5-13, 5-14, 5-15
DMA differences (SHARC processors), 5-6, 5-10, 5-13
DMA (TigerSHARC processors)
  controller, 7-2
  register, 7-2
  transfers, 7-2, 7-4
-dmawidth #, loader switch for Blackfin, 2-9, 3-61
DMISO bit, 6-9, 6-11
.doj (object) files, A-5
DTYPE register, 4-12, 5-7, 5-11
dual-core architectures, *See* ADSP-BF561 processors
DWARF-2 debugging information, 1-8
.dxe (executable) files, 1-16, 2-7, 3-59, 4-27, 6-43, 7-8, A-5, A-15

E

EBOOT pins
  ADSP-2106x/160 processors, 4-5, 4-7, 4-11, 4-15
  ADSP-21161 processors, 5-4, 5-5, 5-6, 5-9, 5-13, 5-15, 5-22
ECEP0 register, 4-8, 5-7, 5-8, 5-9, 5-11
ECPP register, 6-6
ECx register, 4-8, 4-9, 4-11, 4-12
-e filename, loader switch for ADSP-2106x/160 processors, 4-28
-efilename, loader switch for SHARC, 5-28
EIEP0 register, 4-8, 5-8, 5-11
EIPP register, 6-6
EIx register, 4-8, 4-9, 4-12
elf2flt utility, B-3
elfloader, *See* loader
ELF to BFLT file converter, B-3
EMEP0 register, 4-8, 5-8, 5-11
EMPP register, 6-6
EMx register, 4-8, 4-9, 4-12
-enc dll_filename, loader switch for Blackfin, 3-61
encryption functions, 3-61, 3-63, 3-67
end-of-file records, A-8
EP0I vector, 4-13, 5-9, 5-12
EPB0 buffer, 4-11, 4-12
EPROM boot mode (SHARC processors)
  ADSP-2106x/160 processors, 4-2, 4-5, 4-7, 4-9, 4-11, 4-23, 4-24
  ADSP-21161 processors, 5-2, 5-4, 5-5
  multiprocessor systems, 5-21, 5-22
EPROM/flash boot mode (TigerSHARC processors), 7-2, 7-3, 7-10
EPROM flash memory devices, 1-13
executable and linkable format (ELF)
  executable files (.dxe), 1-2, 1-8, 1-11, A-5
  object files (.doj), A-5
  reference information, A-16
  to binary flat format (BFLT) converter, B-3
external
  memory boot, 1-10
  resistors, 4-8, 7-3
  vector tables, 4-22
external bus interface unit (EBIU), 3-34
external memory (Blackfin processors)
  ADSP-BF531/2/3/4/6/7/8/9 processors, 3-3, 3-4, 3-46
  ADSP-BF535 processors, 3-21, 3-24, 3-25, 3-34
  ADSP-BF561 processors, 3-43, 3-46
  multiprocessor support, 3-46

external memory (SHARC processors)
ADSP-2106x/160 processors, 4-5, 4-6, 4-9, 4-10, 4-14, 4-16, 4-19, 4-22, 4-31
ADSP-21161 processors, 5-4, 5-16, 5-21, 5-31
ADSP-2126x/36x/37x/46x processors, 6-7, 6-22
ADSP-2126x/36x/37x processors), 6-25
external ports (SHARC processors)
ADSP-2106x/160 processors, 4-5, 4-7, 4-9, 4-11, 4-13, 4-14, 4-15, 4-19, 4-24
ADSP-21161 processors, 5-4, 5-5, 5-6, 5-7, 5-8, 5-9, 5-10, 5-12, 5-22
external ports (TigerSHARC processors), 7-2
external vector tables, 5-21
EZ-KIT Lite board targets, 1-10

F

-f h|s1|s2|s3|b, splitter switch, 8-6
-f hex|ascii|binary|include, loader switch for Blackfin, 2-9, 3-62
-fhex|ASCII|binary|include|s1|s2|s3, loader switch for SHARC, 4-29, 5-28, 6-45
-fhex|ascii|binary|s1|s2|s3, loader switch for TigerSHARC, 7-9
file formats
list of, 2-7, 3-59
ASCII, 2-9, 3-62, 7-9, A-14
binary, 2-9, 3-62, 7-9
build files, A-4
byte-stacked (.stk), 8-4, 8-6, 8-7
debugger input files, A-15
hexadecimal (Intel hex-32), 2-9, 3-62, 7-9, 8-4, 8-6
include, 2-9, 3-62, 7-9
reference information, A-16
s-record (Motorola), 7-9, 8-4, 8-6
file formatting
selecting for output, 2-12, 3-63
specifying word width, 3-69
file search rules, 1-17
final blocks
See also last blocks (Blackfin processors)
introduction to, 1-15
SHARC processors, 4-18, 6-18, 6-28
FLAG pins, ADSP-2106x/160 processors, 4-23
flag words (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors, 3-11
ADSP-BF535 processors, 3-33
ADSP-BF561 processors, 3-39, 3-43
flash memory
See also PROM/flash boot mode
ADSP-BF51x processors, 2-3
ADSP-BF52x/54x processors, 2-4
devices, 1-9
hold-time cycle selection, 3-62, 3-74
FLG0 signal, 6-10, 6-11
fltdump utility, B-4
frequency, 4-15, 5-13

G

-ghc #, loader switch for Blackfin, 3-62
global header cookies (Blackfin processors), 3-62
global headers (Blackfin processors)
ADSP-BF535 processors, 3-31, 3-32
ADSP-BF561 processors, 3-38
GPEP0 register, 5-8, 5-11
GPLB0 register, 5-14
GPSRX register, 5-16

H

-h|help
loader switch for Blackfin, 2-9, 3-62
loader switch for SHARC, 4-29, 5-29, 6-45
loader switch for TigerSHARC, 7-9
HBG pin, 4-13
HBR pin, 5-12
HBW bits, 4-12

header files (.h), A-3
See also global headers
header records
byte-stacked format (.stk), A-13
s-record format (.s_#), A-10
hexadecimal format, *See* .h_# (Intel hex-32) file format
hexutil utility, B-2
.h_# (Intel hex-32) file format, 7-9, 8-4, 8-6, A-6, A-12
-HoldTime #, loader switch for Blackfin, 3-62
hold time cycles, 3-24, 3-38
host boot mode, introduction to, 1-14
host boot mode (SHARC processors)
ADSP-2106x/160 processors, 4-2, 4-6, 4-11, 4-13, 4-24
ADSP-21161 processors, 5-2, 5-9
ADSP-2126x/36x/37x/46x processors, 6-7, 6-10, 6-17
host boot mode (TigerSHARC processors), 7-2, 7-4, 7-9
host DMA boot mode, ADSP-BF52x/54x processors, 2-4
-hostwidth #, loader switch for SHARC, 5-29, 6-15, 6-26, 6-45
HPM bit, 4-12

I

ICPP register, 6-6
-id#exe=filename
loader switch for SHARC, 4-24, 4-29, 5-23, 5-29, 6-45
loader switch for TigerSHARC, 7-6, 7-9
-id#exe=N, loader switch for SHARC, 5-29
IDLE instruction, 4-4, 4-14, 4-19, 4-20, 5-6, 5-10, 5-13, 5-15, 6-22
idle state, 3-38, 7-3
-id#ref=N, loader switch for SHARC, 4-29, 6-46
ignore blocks (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors, 3-12
ADSP-BF561 processors, 3-39
IIEP0 register, 4-8, 5-8, 5-11
IILB0 register, 5-14
IIPP register, 6-6
IISPI register, 6-9, 6-11
IISRX register, 5-16
IIVT bit, 4-22, 5-21, 6-22
IIx register, 4-8, 4-9, 4-15
image files, *See* PROM, non-bootable files
IMASK register, 4-13, 4-14, 4-16
IMDW register, 4-14, 6-30
IMEP0 register, 4-8, 5-8, 5-11
IMLB0 register, 5-14
IMPP register, 6-6
IMSPI register, 6-9, 6-11
IMSRX register, 5-16
IMx register, 4-8, 4-9
include file format, 7-9, A-8
-initcall, ADSP-BF52x/54x Blackfin loader switch, 2-11
-init filename, loader switch for Blackfin, 2-10, 2-14, 3-11, 3-47, 3-55, 3-62, 3-66
initialization
external memory, 7-10
file inclusion, 2-10, 2-20, 3-62, 3-74
initialization blocks
(ADSP-2126x/36x/37x/46x processors), 6-24, 6-26, 6-27, 6-28, 6-30
initialization blocks (Blackfin processors), 2-17
ADSP-BF531/2/3/4/6/7/8/9 processors, 3-11, 3-13, 3-48
ADSP-BF561 processors, 3-43, 3-44, 3-48
code example, 3-14, 3-48
initialization calls, 2-11
initial word option (SHARC processors), 6-14, 6-15
INIT_L16 blocks, 6-27
INIT_L48 blocks, 6-26

INIT_L64 blocks, 6-28
input file formats, *See* source file formats
input files
  executable (.dxe) files, 2-7, 3-59, 4-26, 5-25, 6-42, 7-8
  extracting memory sections from, 8-5, 8-6
  in multiprocessor systems, 7-6
instruction SRAM (Blackfin processors)
  ADSP-BF531/2/3/4/6/7/8/9 processors, 3-19
  ADSP-BF535 processors, 3-33
  ADSP-BF561 processors, 3-44, 3-45
Intel hex-32 file format, 2-9, 3-62, 7-9, A-6
internal boot mode (SHARC processors), 6-17
internal memory, boot-loadable file execution, 1-10
internal vector tables, 4-22, 5-21
interrupt vector location, 5-9, 5-12
interrupt vector tables, 4-22, 5-21, 6-17, 6-18, 6-22, 6-29, 6-30
IOP registers, 4-12, 4-13
IRQ vector, 4-8, 4-9
IVG15 lowest priority interrupt, 3-7, 3-12, 3-23, 3-37

K

-kb prom|flash|spi|spislave|UART|TWI|FIFO, loader switch for Blackfin, 3-63
-kb prom|flash|spi|spislave|uart|twi|fifo|otp|nand, loader switch for Blackfin, 2-12
-kenc dll_filename, loader switch for Blackfin, 3-63
Kernel (Load) page (Blackfin processors), 2-20, 3-76
kernels (ADSP-2106x/160 processors)
  boot sequence, 4-3, 4-16
  default source files, 4-17, 4-21
  loading to processor, 4-10, 4-13
  modifying, 4-19
  rebuilding, 4-22
  replacing with application code, 4-18
  specifying user kernel, 4-30
kernels (ADSP-21161 processors)
  boot sequence, 5-3
  default source files, 5-16, 5-19
  modifying, 5-18, 5-19
  rebuilding, 5-18, 5-19
kernels (ADSP-2126x/36x/37x/46x processors)
  boot sequence, 6-3, 6-19
  compression/decompression, 6-35, 6-36, 6-40
  default source files, 6-19
  loading to processor, 6-8, 6-13
  modifying, 6-20
  omitting in output, 6-17
  rebuilding, 6-20, 6-21
kernels (Blackfin processors)
  See also second-stage loader
  compression/decompression, 3-50, 3-55
  graphical user interface, 2-20, 3-76
  omitting in output, 3-65
  specifying boot mode, 2-12, 2-19, 3-63, 3-73
  specifying file format, 2-12, 2-13, 3-63
  specifying file width, 2-20, 3-64, 3-73
  specifying hex address, 2-12, 3-63
  specifying hold time, 3-74
  specifying kernel and app files, 2-22, 3-77
  specifying user kernel, 2-13, 3-64
kernels (TigerSHARC processors)
  modifying, 7-5
  omitting in output, 7-4, 7-10
  source files, 7-4
  specifying user kernel, 7-10

-kf hex|ascii|binary|include, loader switch for Blackfin, 2-12, 3-63
.knl (kernel code) files, 2-7, 3-59
-kp #, loader switch for Blackfin, 2-12, 2-14, 3-63, 3-66
-kWidth #, loader switch for Blackfin, 2-13, 3-64

L

L1 memory (Blackfin processors)
  ADSP-BF531/2/3/4/6/7/8/9 processors, 3-7, 3-12, 3-19
  ADSP-BF535 processors, 3-24, 3-33, 3-34
  ADSP-BF561 processors, 3-37, 3-45
L2 memory (Blackfin processors)
  ADSP-BF535 processors, 3-23, 3-24, 3-33, 3-34
  ADSP-BF561 processors, 3-44, 3-45
last blocks (Blackfin processors)
  ADSP-BF531/2/3/4/6/7/8/9 processors, 3-12, 3-13
  ADSP-BF561 processors, 3-39
LBOOT pins
  ADSP-2106x/161 processors, 4-5, 4-7, 4-11
  ADSP-21161 processors, 5-4, 5-5, 5-6, 5-9, 5-13, 5-15
LCOM register, 4-15
LCTL register, 4-15, 4-19, 5-13
.ldr (loader output) files
  ASCII format, A-4, A-14
  binary format, A-9
  hex-32 format, A-6
  include format files, A-8
  naming, 2-14, 3-66, 7-10
  specifying host bus width, 5-29, 6-45
least significant bit first (LSB) format, 6-12
library files (.dlb), A-5
link buffers, 4-15, 5-12, 5-13
linker
  command-line files (.txt), A-4
  description file (LDF) *See* .ldf files
  memory map files (.map), A-6
  output files (.dxe, .sm, .ovl), 1-8, A-5
linking, introduction to, 1-8
link port boot mode (SHARC processors)
  ADSP-2106x/160 processors, 4-2, 4-5, 4-15
  ADSP-21161 processors, 5-2, 5-4, 5-12
link port boot mode (TigerSHARC processors), 7-3, 7-9
loadable files, *See* boot-loadable files
loader
  operations, 1-11
  output file formats, 1-11, 1-16, A-6, A-8, A-9
  setting options, 2-18, 3-72, 4-32, 5-32, 6-49, 7-12, 8-9
loader file formats (ADSP-BF535 processors)
  PROM/flash boot with kernel, 3-30
  PROM/flash boot without kernel, 3-29
  PROM/flash/SPI boot with kernel, 3-28, 3-30
loader for ADSP-2106x/21160 processors, 4-1
loader for ADSP-21161 processors, 5-1
loader for ADSP-2126x/36x/37x/469 processors, 6-1
loader for ADSP-BF51x/52x/54x Blackfin (includes splitter), 2-1
loader for ADSP-BF53x/BF561 Blackfin (includes splitter), 3-1
loader for Blackfin (includes splitter)
  command-line syntax, 2-6, 2-8, 3-58, 3-60
  default settings, 2-18, 3-72
  graphical user interface, 2-19, 3-72
  list of switches, 2-8, 3-60
loader for TigerSHARC
  command-line syntax, 7-6
  graphical user interface, 7-12
  list of switches, 7-9
  operations, 7-1

loader kernels, *See* boot kernels
loader output, *See* output files
loader switches, *See* switches by name
loading, introduction to, 1-9
Load page
  Blackfin processors, 2-18, 3-72
  SHARC processors, 4-32, 5-32, 6-49
  TigerSHARC processors, 7-12
Load (Splitter) page (Blackfin processors), 2-22, 3-78
-l userkernel
  loader switch for Blackfin, 2-13, 3-46, 3-64
  loader switch for SHARC, 4-30, 5-30, 6-20, 6-40, 6-46
  loader switch for TigerSHARC, 7-9, 7-10

M

-M, loader switch for Blackfin, 2-13, 2-14, 3-64, 3-65
make files, 2-13, 2-14, 3-64, 3-65
.map (memory map) files, A-6
-maskaddr #, loader switch for Blackfin, 2-13, 3-64
masking EPROM address bits, 2-13, 3-64
master (host) boot, introduction to, 1-10
-MaxBlockSize #, loader switch for Blackfin, 2-13, 3-64
-MaxFillBlockSize #, loader switch for Blackfin, 2-13
-MaxZeroFillBlockSize #, loader switch for Blackfin, 3-65
memory map files (.map), A-6
memory ranges (Blackfin processors)
  ADSP-BF531/2/3/4/6/7/8/9 processors, 3-19
  ADSP-BF535 processors, 3-33
  ADSP-BF561 processors, 3-45
microcontroller data transfers, A-12
-MM, loader switch for Blackfin, 2-14, 3-65
MODE1 register, 4-13
MODE2 register, 4-13, 4-14
-Mo filename, loader switch for Blackfin, 2-14, 3-65
most significant bit first (MSB) format, 6-12
Motorola S-record file format, 7-9, A-10
MSBF bit, 6-9, 6-10
MS bit, 6-9, 6-10
MSWF register, 5-7, 5-11
-Mt filename, loader switch for Blackfin, 2-14, 3-65
Multi, 2-17
multiprocessor booting, introduction to, 1-10
multiprocessor systems (Blackfin processors), 2-17, 3-47
  See also dual-core systems
multiprocessor systems (SHARC processors)
  ADSP-2106x/21160 processors, 4-23, 4-24
  ADSP-21161 processors, 5-7, 5-21, 5-22, 5-23
  ADSP-2136x/37x processors, 6-33
multiprocessor systems (TigerSHARC processors), 7-6, 7-9

N

NAND flash boot mode, ADSP-BF52x/54x processors, 2-4
-no2kernel, loader switch for Blackfin, 3-65
no-boot mode
  introduction to, 1-10, 1-13
  See also internal boot mode
  selecting with -romsplitter switch, 2-15, 3-67
no-boot mode (Blackfin processors)
  ADSP-BF51x processors, 2-3
  ADSP-BF52x/54x processors, 2-4
  ADSP-BF531/2/3/4/6/7/8/9 processors, 3-4
  ADSP-BF535 processors, 3-21
  ADSP-BF561 processors, 3-37
  selecting, 2-19, 3-16, 3-73, 3-79
no-boot mode (SHARC processors)
  ADSP-2106x/160 processors, 4-2, 4-6, 4-16
  ADSP-21161 processors, 5-2, 5-4, 5-16

-nofinalblock, loader switch for Blackfin, 3-65
-nofinaltag, loader switch for Blackfin, 3-65
-noinitcode, loader switch for Blackfin, 2-14, 3-66
-nokernel
  loader switch for ADSP-2126x/36x/37x/46x processors, 6-46
  loader switch for Blackfin, 3-66
  loader switch for SHARC, 6-17
  loader switch for TigerSHARC, 7-4, 7-10
non-bootable files
  introduction to, 1-10, 1-15
  creating from command line, 8-2
  creating from IDDE, 8-9
  ignoring ROM sections, 8-5
  specifying format, 8-6
  specifying name, 8-5
  specifying word width, 8-4, 8-6
NOP instruction, 4-4, 4-14, 4-19, 4-20, 5-6, 5-10, 5-13, 5-15, 6-22
-norom, splitter switch, 8-5
-nosecondstageloader, loader switch for Blackfin, 3-66
-nozeroblock, loader switch for SHARC, 5-30, 6-47
numeric formats, A-3

O

-o2, loader switch for Blackfin, 2-12, 2-14, 3-63, 3-66
object files (.doj), A-5
-o filename
  loader switch for Blackfin, 2-14, 3-66
  loader switch for SHARC, 4-30, 5-30, 6-47
  loader switch for TigerSHARC, 7-10
  splitter switch, 8-5
on-chip boot ROM
  introduction to, 1-14
  ADSP-BF531/2/3/4/6/7/8/9 processors, 1-15, 3-3, 3-7, 3-9, 3-11, 3-19, 3-47
  ADSP-BF535 processors, 3-21, 3-23, 3-25
  ADSP-BF561 processors, 3-35, 3-37, 3-43, 3-44, 3-45, 3-47
OTP boot mode, ADSP-BF51x processors, 2-3
OTP boot mode, ADSP-BF52x/54x processors, 2-4
output files
  See also -o loader switch
  generating kernel and application, 2-14, 3-66
  specifying format, 1-12, A-5
  specifying name, 2-14, 3-66, 7-10
  specifying with -o switch, B-2
  specifying word width, 3-69, 5-29
overlay compression, 6-39
overlay memory files (.ovl), 2-7, 3-59, 7-8, A-5, A-15

P

-p #
  loader switch for Blackfin, 2-14, 3-66
  loader switch for TigerSHARC, 7-10
packing boot data, 5-2, 7-2
-paddress, loader switch for SHARC, 4-30, 5-30, 6-47
parallel ports, 6-6
parallel/serial PROM devices, 1-14
-pflag #|PF|PG|PH #, loader switch for Blackfin, 3-67, 3-69, 3-70, 3-71, 3-74
PFx signals, 3-67
placement rules, of the command-line, 2-6, 3-58, 7-6
PMODE register, 4-9, 4-12, 5-7, 5-11
-pm splitter switch, 8-5
PP16 bit, 6-6
PPALEPL bit, 6-6
PPBHC bit, 6-6

PPBHD bit, 6-6
PPCTL register, 6-5, 6-6
PPDEN bit, 6-6
PPDUR bit, 6-6
PPEN bit, 6-6
PPTRAN bit, 6-6
processor IDs, 4-23, 4-24, 5-22, 5-23, 7-6, 7-9
  assigning to .dxe file, 4-29, 5-29, 6-45
  pointing to jump table, 4-29, 5-29
processor-loadable files, introduction to, 1-13
processor type bits (Blackfin boot streams), 3-11
-proc part_number
  loader switch for Blackfin, 2-15, 3-67
  loader switch for SHARC, 4-30, 5-30, 6-47
  loader switch for TigerSHARC, 7-9, 7-10
  splitter switch, 8-7
program counter settings (ADSP-2106x/160 processors), 4-12
program development flow, 1-7
program memory sections (splitter), 8-3, 8-5
Project Options dialog box, 1-12, 2-5, 2-18, 2-19, 3-57, 3-72, 3-73, 6-20, 7-4, 7-5
PROM
  boot mode, introduction to, 1-14
  downloading boot-loadable files, 1-10
  memory devices, 6-16, A-6
PROM boot mode, ADSP-2126x/36x/37x/46x processors, 6-5, 6-21, 6-29
PROM boot mode, TigerSHARC processors, 7-4, 7-9
PROM/flash boot mode (Blackfin processors)
  ADSP-535 processors, 3-62
  ADSP-BF531/2/3/4/6/7/8/9 processors, 3-4, 3-48
  ADSP-BF535 processors, 3-21, 3-28, 3-29, 3-34
  ADSP-BF561 processors, 3-43, 3-48
PROM (image) files
  creating from command line, 8-2
  creating from GUI, 8-9
  ignoring ROM sections, 8-5
  specifying format, 8-6
  specifying name, 8-5
  specifying width, 8-6
pull-up resistors, 5-7
Px register, 4-19, 6-30

Q

-quickboot, loader switch for Blackfin, 2-15

R

-r #, splitter switch, 8-6
-ram, splitter switch, 8-5, 8-6
RBAM bit, 5-7
RBWS bit, 5-7
RD pin, 4-10, 5-8
-readall, loader switch for Blackfin, 2-15
references, file formats, A-16
RESET
  interrupt service routine, 3-7, 3-23, 3-37, 5-12
  pin, 4-9, 5-8, 5-11, 7-3

reset
  processor, introduction to, 1-14, 1-15
  ADSP-2106x/160 processors, 4-3, 4-8, 4-11, 4-12, 4-16
  ADSP-21161 processors, 5-3, 5-6, 5-7, 5-9, 5-10, 5-13, 5-15
  ADSP-2126x/36x/37x/46x processors, 6-3, 6-6, 6-9, 6-10, 6-17, 6-20
  ADSP-BF561 processors, 3-16, 3-35, 3-37, 3-79
  Blackfin processors, 2-2, 3-2, 3-3, 3-21, 3-25
  dual-core Blackfin processors, 3-35
  TigerSHARC processors, 7-2, 7-3, 7-4
  vector addresses, 4-4, 4-9, 4-14, 5-20, 6-21
  vector routine, 3-18, 3-81, 5-9
resistors (pull-up), 7-3
restrictions, second-stage loader, 3-34
-retainSecondStageKernel, loader switch for SHARC, 6-47
ROM
  memory images as ASCII text files, A-14
  memory sections, 8-5
  setting splitter options (Blackfin processors), 2-22, 3-78
  splitter, *See* splitter
-romsplitter, loader switch for Blackfin, 2-13, 2-15, 3-64, 3-67
Rx registers, 3-43, 3-48, 4-13
RXSPI register, 6-8
RXSR register, 6-8
RXx registers, 6-9

S

s1 (Motorola EXORciser) file format, 7-9, 8-6, A-10
s2 (Motorola EXORMAX) file format, 7-9, 8-6, A-10
s3 (Motorola 32-bit) file format, 7-9, 8-6, A-10
-save section, loader switch for Blackfin, 2-16
scratchpad memory (Blackfin processors)
  ADSP-BF535 processors, 3-33
  ADSP-BF561 processors, 3-45
SDCTL register, 5-18, 6-20
SDRAM/DDR boot mode, ADSP-BF52x/54x processors, 2-4
SDRAM memory (ADSP-2106x/160 processors), 4-17
SDRAM memory (Blackfin processors)
  ADSP-BF531/2/3/4/6/7/8/9 processors, 3-11, 3-15, 3-20
  ADSP-BF535 processors, 3-24, 3-33, 3-34
  ADSP-BF561 processors, 3-43, 3-45
SDRDIV register, 5-18, 6-20
second-stage loader
  ADSP-BF535 processors, 1-15, 3-24, 3-27, 3-32, 3-33, 3-34
  ADSP-BF561 processors, 3-43, 3-44
  creating from VisualDSP++, 2-21, 3-77
  setting options, 2-19, 3-73, 3-75
  source files (ADSP-BF535 processors), 3-77
SENDZ bit, 6-9, 6-11
sequential EPROM boot, 5-22
shared memory
  Blackfin processors, 3-44, 3-45
  file format (.sm), 2-7, 3-44, 3-59, 7-8, A-5, A-15
  in compressed .ldr files, 6-36, 6-39
  omitting from loader file, 4-28, 5-28
shift register, *See* RX registers
-ShowEncryptionMessage, loader switch for Blackfin, 3-67
silicon revision, setting, 2-16, 3-68, 4-31, 5-31, 6-48, 7-11, 8-8
simulators, for boot simulation, 1-10
single-processor systems, 4-24, 5-23, 7-6, 7-9, 8-2

-si-revision #|none|any
  loader switch for Blackfin, 2-16, 3-68
  loader switch for SHARC, 4-31, 5-31, 6-48
  loader switch for TigerSHARC, 7-11
  splitter switch, 8-8
slave processors, 1-10, 1-14, 6-10
.s_# (Motorola S-record) files, 8-4, A-10
.sm (shared memory) files, 2-7, 3-59, 4-28, 5-28, 7-8, A-5, A-15
software reset, 1-13, 3-7, 3-23, 3-37
source file formats
  assembly text (.asm), A-2
  C/C++ text (.c, .cpp, .cxx), A-2
SPIBAUD register, 6-11
SPI boot modes (SHARC processors)
  ADSP-21161 processors, 5-2, 5-4, 5-14
  ADSP-2126x/36x/37x/46x processors, 6-7, 6-13, 6-21, 6-29
SPICLK register, 6-8, 6-10, 6-13, 6-17
SPICTL register, 5-15, 6-9, 6-11
SPIDMAC register, 6-9, 6-11
SPIDS signal, 6-8
SPI EEPROM boot mode (Blackfin processors)
  ADSP-BF535 processors, 3-21, 3-24, 3-28, 3-29
  ADSP-BF561 processors, 3-43
SPIEN bit, 6-9, 6-10
SPI_FLAG0_O signal, 6-10, 6-11
SPI flash boot mode (ADSP-2126x/2136x/2137x/21469 processors), 6-16
SPIFLG register, 6-11
SPI host boot mode (ADSP-2126x/36x/37x/46x processors), 6-17
SPI master boot modes
  ADSP-2126x/36x/37x/46x processors, 6-7, 6-10, 6-14
  ADSP-2126x/36x/37x processors, 6-18
  ADSP-BF51x processors, 2-3
  ADSP-BF52x/54x processors, 2-4
  ADSP-BF531/2/3/8/9 processors, 3-4
  ADSP-BF534/6/7 processors, 2-4, 3-4
  See also SPI flash, SPI ROM, host processor master boot modes
SPI memory slave devices, 6-11
SPI PROM boot mode (ADSP-2126x/36x/37x/46x processors), 6-13, 6-14, 6-16
SPIRCV bit, 6-9, 6-11
SPIRx register, 5-2, 5-14, 5-15
SPI slave boot mode (ADSP-2126x/2136x/2137x/21469 processors), 6-8
SPI slave boot mode (ADSP-2126x/36x/37x/46x processors), 6-7, 6-14
SPI slave boot mode (Blackfin processors)
  ADSP-BF51x processors, 2-3
  ADSP-BF52x/54x processors, 2-4
  ADSP-BF531/2/3/8/9 processors, 3-4
  ADSP-BF534/6/7 processors, 3-4
Split page, 8-9
splitter
  introduction to, 1-9, 1-11, 1-12, 1-14
  as ROM splitter on Blackfin processors, 2-19, 3-73
  command-line syntax, 8-2
  file extensions, 8-4
  graphical user interface, 8-9
  list of switches, 8-5
  output file formats, A-10, A-12, A-14
SPORT hex data files, A-15

SRAM memory (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors, 3-19
ADSP-BF535 processors, 3-33
ADSP-BF561 processors, 3-37, 3-45
-s section_name, splitter switch, 8-6
start addresses
ADSP-2106x/160 application code, 4-4
Blackfin application code, 2-14, 2-20, 3-66, 3-74
status information, 2-16, 2-20, 3-68, 3-74
.stk (byte-stacked) files, 8-4, 8-6, 8-7, A-12
streams, *See* boot streams
supervisor mode (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors, 3-7
ADSP-BF535 processors, 3-23
ADSP-BF561 processors, 3-37
synchronous boot operations, 4-13
SYSCON register (SHARC processors)
ADSP-2106x/160 processors, 4-10, 4-12, 4-13, 4-19, 4-22
ADSP-21161 processors, 5-18, 5-21
ADSP-2126x/36x/37x/46x processors, 6-20, 6-22
SYSCR register (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors, 3-8
ADSP-BF561 processors, 3-37, 3-38
SYSCTL register, 6-31
SYSTAT register, 4-23
system reset configuration register, *See* SYSCR register

T

-t#
loader switch for SHARC, 4-31, 5-31
loader switch for TigerSHARC, 7-10
termination records, A-11
text files, A-4, A-14
TigerSHARC processors, boot modes, 7-2, 7-3, 7-9
timeout cycles (TigerSHARC processors), 7-10
two-wire interface (TWI) boot mode
ADSP-BF2x/54x processors, 2-4
ADSP-BF534/6/7 processors, 3-4, 3-12
.txt (ASCII text) files, A-4

U

-u, splitter switch, 8-7
UART slave boot mode (Blackfin processors), 2-3, 2-4, 3-4
UBWM register, 4-11
uncompressed streams, 3-53, 6-38
-use32bitTagsforExternal Memory Blocks, loader switch for SHARC, 4-31
utility programs, B-1

V

.VAR directive, A-3
vector addresses, 4-20, 5-20
-version
loader switch for SHARC, 4-32, 5-32, 6-48
loader switch for TigerSHARC, 7-11
splitter switch, 8-8
-v (verbose)
loader switch for Blackfin, 2-16, 3-68
loader switch for SHARC, 4-32, 5-32, 6-48
loader switch for TigerSHARC, 7-10

W

WAIT register, 4-9, 4-11, 4-17, 4-19, 5-7, 5-18, 6-20
-waits #, loader switch for Blackfin, 3-68
wait states, 3-24, 3-68, 3-73, 4-10, 4-15, 5-7, 5-8
-width #, loader switch for Blackfin, 2-16, 3-64, 3-69
WIDTH() command, 6-7
WL bit, 6-9, 6-10

word width, setting for loader output file, 5-29, 6-45

Z

zero-fill blocks (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors, 3-11, 3-65
ADSP-BF561 processors, 3-39
zero-fill blocks (SHARC processors)
ADSP-2106x/160 processors, 4-18
ADSP-2126x/36x/37x processors, 6-25
zero-padding (ADSP-2126x/36x/37x/46x processors), 6-26, 6-27
-zeroPadForced #, loader switch for Blackfin, 3-69